Baby Steps

How To Become An Excel God Without Really Trying

Alexander J Turner

Dedicated to: Professor I. H. Williams.

For his tireless support and guidance of myself and many others in our search for a greater understanding of nature's creation.

Table of Contents

Forward: Run The World From Excel!

Most calculations and nearly all business processes start out life in Excel. They grown in complexity and slowly require multiple spreadsheets to function. Human error leaks into the mix and the whole house of cards comes crashing down.

The standard solution to this challenge is to commission some huge software system to "solve all our problems in one hit". This super system *will* be delivered late and *will* be over budget. What is worse, the requirements will have moved on since the system was designed. The wonderful (wonderfully expensive) system will solve yesterday's problems and not address today's.

The final act in our IT tragedy is where the inadequacies of the super system are overcome by pre and post processing all the data with Excel!

All the above pain can so easily be avoided. It is caused by Excel only "going so far". Excel will process data, but not automatically chain processing steps together. This lack of chaining means that humans end up in the processes. Even highly trained humans make mistakes at the rate of 1 in 200 to 300 hundred repeated actions. Data processing which involve significant human interaction will become unstuck irrespective of how hard the people try.

A Dream – No Reality

How about a world where data processing and business decisions take place in Excel without human error? All repetitive tasks are automated and efficient. In this world, we would have the flexibility and responsiveness of Excel but the reliability and speed of a dedicated system.

How about adding to this wish list? It would also be great if even the most complex processes were rendered so simple to set up that non programmers could do it!

This is not a fantasy world, it exists. The techniques of Excel Scripting Macros do exactly what is required. I believe that an entire business, of thousands of people, could be run with just Excel Scripting Macros, Excel and a free database. 90% of all business processes can be performed even without the database, all we need is Excel and Scripting Macros.

Who can benefit from Excel Scripting Macros?

- ✔ If you are in a small business and want to automated work flow – this is for you.
- ✔ If you are in a huge business and want to process performance figures – this is for you.
- ✔ If you are researching animal behaviour and need to analyse results – this is for you.
- ✔ If you want to apply a rule set to choose who, out of 10 000 employees, gets a Christmas bonus – this is for you.
- ✔ If you want to do just about anything that involved numbers, data or rules – this is for you!

Where Does This Idea Come From?

The school of hard kicks, that this where is comes from. Back in 2001-2002 I began to realise two things. Firstly, the huge and complex Java based systems on which I was working just never provided what real people wanted. Secondly, nearly all business was already being performed in Excel.

It was at this time that I set up a company 'JDIT' to take advantage of these realisations. We took Excel and, using Excel macros, pushed way beyond what most people had dreamt of. However, despite the success of the concept, it was not fully what was needed. The systems JDIT produced were too large and two complex. In some ways, they turned Excel into the huge and complex systems I was trying to get away from.

The key breakthrough came when considering how to directly interconnect Excel with a large web based system for which I was architect. I realised that a simple script could run Excel and place data into it. The technology I had used in JDIT “inside Excel” could be put “outside Excel” in a script. However, the result was much more powerful than I had expected because it allowed complex things to be done in simple ways. Contrast this, if you would, with normal IT where simple things are done in extremely complex ways!

Excel Scripting Macros evolved a fair way from this initial “wow” moment. The next challenge was to beat every last ounce of complexity out of the approach. That was not as straight forward as one might imagine. But, each time I found an approach which reduced complexity, the project on which I was working benefited enormously. In short, I knew I was on the right track.

Why 'Baby Steps'

Up to this point I had thought of Excel Scripting Macros as a good way to solve problems on the projects with which I was involved . As time went on, feedback from my co-workers and my blog (www.nerds-central.com) started to indicate they had crossed a watershed. No longer was this seen as a programmer concept but more as a business tool. The opportunity to devolve data and decision processing from programmers to the people who needed it was tantalising to say the least.

I knew that to achieve this breakthrough democratisation of technology, the learning curve would have to be very shallow. Hence, I came up with the notion of 'Baby Steps'; A series of tiny steps which take a person from being familiar with Excel through to being able to perform seemingly super human feats with it.

Having considered many formats from blog to podcast, I figured a friendly book format was ideal. A book that is not full of padding to get the page count up. A book that is small enough to fit in a laptop case and read on the train. In short, a book which, like its contents, demands only the minimum of effort invested for the maximum of benefit given.

Introduction: What is this all about then?

'Baby steps' is all about learning how to write 'Scripting Macros' for Excel. I am sure that you have heard of Excel Macros. These are little programs embedded into an Excel Workbook which automate tasks. Scripting macros are very similar. The biggest difference is that they do not 'live' inside a Workbook. They are small files that exist outside Workbooks but can act upon one or more Workbooks at once.

Because Scripting Macros are outside of the Workbooks on which they act, they are very much more flexible than traditional Macros. They also get around many of the nasty security problems and consequent restrictions that are part and parcel of Excel Macros these days.

The concept behind 'Baby Steps' is that, whilst you can be something massively complex with Scripting Macros, it is also possible to do amazing stuff with massively simple Scripting Macros. Experience in the field has shown that the approach has huge benefits to business processes which are dependant on people sending data to one another via Excel spreadsheets. Automation via Scripting Macros removes 'typo' mistakes which people tend to make when faced with repetitive tasks. Additionally, one or two simple Scripting Macros can process data that would traditionally be loaded into a database and from which programmers would write reports.

Do I Need To Be A Programmer?

100% no. The idea behind 'Baby steps' is that a little knowledge will take you a very long way. Each concept is tied back to 'real world' examples and every effort possible has been taken to remove jargon!

If you can use Excel, you can understand 'Baby Steps'.

You do not need to be a programmer.

This book will not try to teach you to be a programmer.

Using This Book

If you have never done any computer scripting at all, do not worry. Everything you need to know is contained in lessons 1,2 and 3. If you are slightly familiar with the concepts of scripts, loops Variables and all that stuff, then dive in at lesson 4.

Even if you are an expert programmer, you may gain a lot from the later chapters. I would recommend a little speed reading at lesson 4 to pick up a few tips and then dive in more at lesson 5.

Lesson 1: Basic Concepts

Let us start off considering drinking a glass of wine...

To instruct a computer that a person should drink a glass of wine, it seems logical to tell the computer that there is a 'thing' called person and a 'thing' called `glass-of-wine`. Once we have established this set-up, we can say something like:

person drink glass-of-wine

(The hyphens are included just for clarity, to group words together)

Now we have started writing a Scripting Macro. To make this a bit more realistic we need to split out the concept of '*a glass of wine*' from '*my glass of wine*'. After all, you want to know which glass your are drinking (at least at the start of the evening).

To do this, we might write something like this:

```
set my-glass-of-wine to be a-glass-of-wine
set my-person to be a-person
my-person.drink(the-glass-of-wine)
```

You can probably tell that I am going somewhere with the layout of these 'pseudo scripts'; I am transforming every day speak into the way we tell the computer to do stuff. A lot of the time, it is fairly straight forward to do so. Sometimes, however, history and convention get in the way. But, for now, let us continue along this path.

Key Concept:
*In computer speak, we call 'my-glass-of-wine' an **Object** and we call 'a-glass-of-wine' a class of objects, or simply a **Class**.*

One really nice thing about computing is that making glasses of wine is very simple and just about free. If we want a new glass of wine, we just say something like 'Create my-glass-of-wine which is from the Class of 'a-glass-of-wine'.

Making glasses of wine in a script might look something like this:

```
set my-glass-of-wine = CreateObject(“a-glass-of-wine”)
```

But actually, hyphens between words are not allowed so we put:

```
set myGlassOfWinne = CreateObject(“aGassOfWine”)
```

That does not look very much like everyday speak any longer, but there is a clear lineage to be seen! One question that comes up is what does the '=' sign mean? There are two things '=' can mean in Scripting Macros:

```
1) Does x equal y, as in:
     if x = y then
2) Make x equal to y, i.e. load the value of y into
   x, as in:
     Set x = y
   or:
     set myExcel=CreateObject("Excel.Application")
```

Now we can put all this together to make a nearly complete program:

```
' Create the Objects which we require
set myself  = CreateObject("People.Person")
set myGlass = CreateObject("Wine.Glass")

' Do stuff with the Objects, in this case
' Object myself, drink the Object myGlass:
' drink is a pre-defined Method which any
' Object from the Class People.Person can do.

mySelf.drink(myGlass)
```

To explain what is described in the comments to the code above: *.drink* is what we call a Method. It is a method of the Object *mySelf*. We do not have to define what *.drink* is because its definition comes along with Class *People.Person*. Because *mySelf* is a Object from *People.Person* (it is a person Object) it will have the Method *drink*.

Methods (doing things) can have parenthesis after them. Inside the parenthesis there can be a list of things the method is to work with. In the above example, the list has one element *myGlass*. Some methods, like the *.Add* method of Excel's *Workbooks* collection, have no elements in the list and so are followed by empty parenthesis as in *.Workbooks.Add()*.

I have called the Classes People.Person and Wine.Glass, with these double names, because that is the Microsoft convention. An Excel application is called Excel.Application. To create one you put *CreateObject("Excel.Application")*.

Now we have figured out how to get 'Cyber-Sloshed', let us see is we can apply the same thing to Excel!

```
set myExcel = CreateObject("Excel.Application")
```

This simple piece of code actually causes Windows to create a completely new Excel application. The 'Variable' myExcel 'refers' to it. So, later in the program, when we want to do something with Excel, we can use the Variable 'myExcel' to pass instructions to our new Excel application.

For our very first 'baby step' towards being an Excel god, we are going to do just that. We are going to create an instance of Excel and do something with it. That something will be simply to make it visible (they are created hidden) and then create a new, empty Workbook inside it.

Please follow these instructions to the letter to create your very first Excel Macro Script:

```
1) Open Notepad
2) Type in:

     set myExcel = CreateObject("Excel.Application")
     myExcel.visible = TRUE
     myExcel.Workbooks.Add()

3) Save your magical words as step1.vbs on your desktop.
4) Double click on the Icon that has appeared on your
desktop to represent you new file.
5) Stand back in amazement!
```

This piece of code can be described in words like this:

> Set the Variable *myExcel* to be an newly created Object which is from the Class *“Excel.Application”*. Make the *visible* Property of that Object be *true*. Get the *Workbooks* Property of that Object and run its Method *Add*, which will cause a new *Workbook* to be created.

'Take Home' Ideas from Lesson 1

We have seen that Macro Scripts can control Excel and that they have a direct lineage from normal English.

When writing scripts, we represent real, tangible things (like glasses of wine or workbooks) as Objects. The object 'myGlassOfWine' comes from the Class of Objects 'aGlassOfWine'.

Macro Scripts are simple text files. They are edited using a text editor (like notepad) and saved with the '.vbs' file extension.

When writing a Scripting Macro, before we do anything with Excel, we need to create an Excel.Application Object.

Lesson 2: Building Block Concepts

Collections:

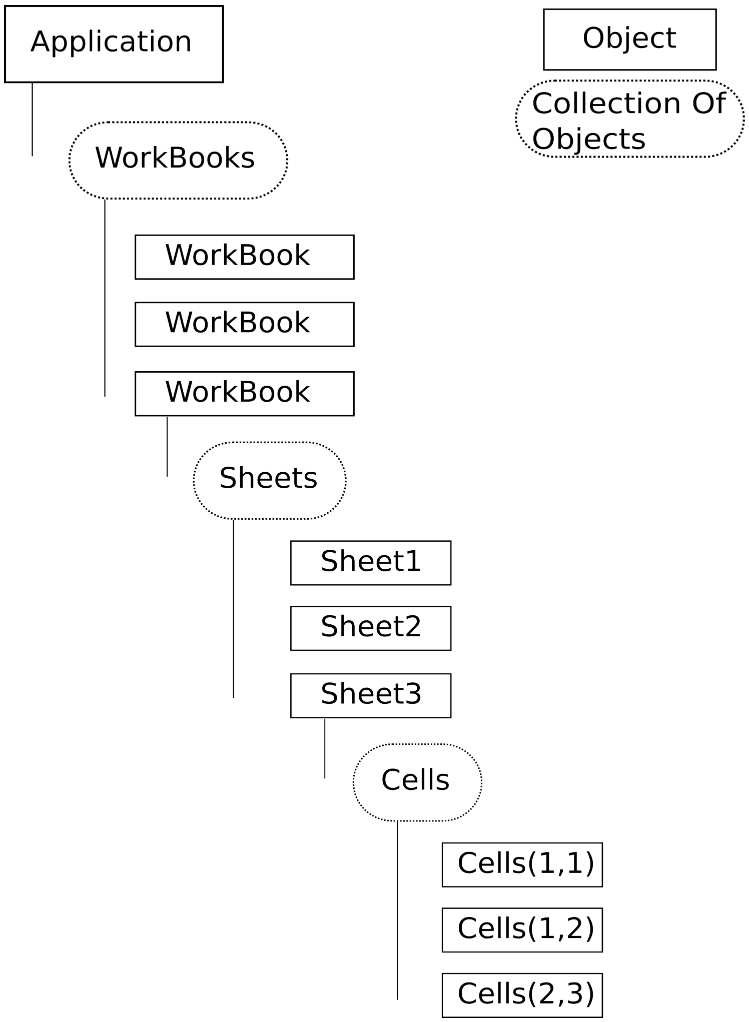


Fig 1: A digram showing how in Excel Objects often have Collections of Objects inside them. A Collection holds a list of zero or more Objects inside it and allows you to find those Objects by index (e.g. 0,1,2 etc., or sometimes: 1,2,3 etc.) or name (e.g. "sheet1","sheet2" etc.). This diagram shows you how to start off with Excel.Application and 'drill' all the way down to individual Cells in a Sheet.

In lesson one we looked at how things, like glasses of wine and Excel Applications, are called Objects. 'myGlassOfWine' is an Object whilst 'aGlassOfWine' defines a Class of Objects and so we call it a Class.

When working with Excel, we use Objects a lot; also, we tend to use a lot of the same Class of Object at once. When you create a *Workbook* Object (like we did with with our simple script) you see three Sheets, not one. Indeed, an Excel.Application can have many *Workbooks* and each *Workbook* can have many Sheets and each *Sheet* will have many *Cells*.

To handle all these groups of Objects, Microsoft use what it calls 'Collections'. These are numbered, and sometimes named, lists of Objects. A Collection is also an Object its self. For example, a Collection of Workbooks is an Object from the Class of Collections.

All Collections have the Property *.count* (note, because it is a Property, not a Method, there is no parenthesis). This tells you how many things are inside the Collection. Some special Collections have other properties and/or Methods. For example, Excel's *Workbooks* Collection has *.Add()* to create a new *Workbook* and *.Open("file name")* to open an existing *Workbook*.

If we draw out the relationships of Excel.Application, Workbook, Sheet and Cell, we get something like Figure 1.

To find the Object for a Cell, we might do something like this:

```
' Create an Excel.Application Object
Set myExcel = CreateObject("Excel.Application")
' Cause Excel to create a Workbook (it starts
' with none). This is done by 'adding' one to
' the Collections of Workbooks
myExcel.Workbooks.Add()

' Find the first Workbook in the Collection
' of Workbooks. i.e. this says: set the
' word 'myWorkbook' to refer to the Workbook
' Object at position 1 in the Collection.
Set myWorkbook = myExcel.Workbooks(1)

' Do the same this time but to get a Sheet
Set mySheet = myWorkbook.Sheets(1)

' Finally, find one particular Cell
Set myCell = mySheet.Cells(1,1)

' Now we have the top left Cell of
' the first Sheet. i.e. the Object which represents
' the Cell at the top left hand corner of the first
' Sheet.
'
' We can use this Object to
' write the value of the contents of the Cell.
myCell.Value = “Hello World”
```

Key Concept:

Collections are a way of storing zero or more Objects so that they can be easily found and used.

Variables:

When we put:

```
Set myExcel= CreateObject(“Excel.Application”)
```

We are saying:

> “Set a Variable called *myExcel* to refer to a newly created Excel.Application Object.”

When working with Objects in Excel, we often set Variables to refer to Objects to make it easy to use these Objects. Here is an example as to why:

```
' Try to set a Cell's contents without using Variables
' Step 1: Open up Excel and create a new Workbook
' in it
CreateObject(“Excel.Application”).Workbooks.Add()
' Are, now I am unstuck, because I no longer have a
' way of 'getting to' to the Excel.Application
' Object.
```

The solution, as we saw before, is to set a Variable to refer to the Excel.Application Object. This approach has other benefits as well. Once we get the hang of it we can:

1. Save our selves a lot of typing.
2. Save the computer a lot of 'thinking'.
3. Make developing (adding to, editing, improving) scripts much easier.

These advantages come from the ability to store 'working' information in Variables just like we might store notes on post-its. If you had to work out what the square root of 23456 was each morning, you might well write down the answer to save the effort! Variables are used for this purpose in scripting.

Key Concept:
Variables are like 'post-it-notes' for scripts; they allow us to store things so we do not have to keep recreating them.

Let us see this 'post-it-note' concept in use by adding to our previous script:

```
' Create an Excel.Application Object
set myExcel = CreateObject(“Excel.Application”)

' Cause Excel to create a Workbook (it starts
' with none). This is done by 'adding' one to
' the Collection of Workbooks
myExcel.Workbooks.Add()

' Find the first Workbook in the
' Collection of Workbooks
set myWorkbook = myExcel.Workbooks(1)

' Do the same but this time to get a Sheet
set mySheet = myWorkbook.Sheets(1)

' Finally, find one particular Cell
set myCell = mySheet.ells(1,1)

' Now we have an Object for the top left Cell of
' the first Sheet. We can use this Object to
' set the value of the contents of the Cell.
myCell.value = “Hello World”

' Now we can re-use the mySheet Variable to get
' the Object for the next Cell in the Column
set myCell   = mySheet.Cells(2,1)
myCell.value = “Hello Again”
```

Here we have re-used the value of mySheet rather than having to re-find the Sheet Object. This use of a Variable has saved us re-wring and re-running all the code which finds the Sheet.

More On Variables

Variables do not only refer to Objects; they can hold other stuff like Numbers, Dates and Strings (pieces of text are called 'Strings' as in 'a string of characters').

When working with Excel Scripting Macros we are using a programming language called VBScript. VBScript, just like any programming language that has been around for a while, has history; it has accumulated a few oddities. Whilst these can be explained, it is often better to just accept them. One of these is the way we work with Variables and things which are not Objects.

Strings, Numbers, Dates and Booleans (logical values, true/false) are not Objects in VBScript. This is not for any particular reason other than history. Variables are almost exactly the same when working with Objects and non Objects, appart from one little thing: we do not 'set' Variables to non objects. So:

```
Set Cell     = mySheet.Cells(1,1) ' OK
myString     = “Hello World”      ' OK
myNumber     = 1.0 ' OK
Set myNumber = 2   ' WRONG (2 is not an object)
myObject = CreateObject(“Excel.Application”) ' WRONG
```

One way of thinking about this is to say that Numbers, Strings, Booleans and Dates are too simple to be thought of as Objects and so are treated differently. Also note that if number is in quotes it is used as text. “1.0” is not a number, it is a String.

You might have also realised by now that any text after ' on a line is considered to be a comment. Comments are not read by the computer and have no effect on the script, they are there to help people reading the script to know what is going on. The comment goes from the first ' to the end of the line.

Now we have a really powerful tool because we can let a Variable read the value from a Cell and we can write a Variable's value into a Cell. This is the basis of most Excel Macro Scripting data processing.

```
' Script to show how values can be read and
' written in and out of Cells and Variables
' =========================================
Set myExcel = CreateObject(“Excel.Application”)
myExcel.Workbooks.Add()
Set myWorkbook = myExcel.Workbooks(1)
Set mySheet = myWorkbook.Sheets(1)

' Start the process by writing a value into
' the top left Cell
Set myCell   = mySheet.Cells(1,1)
myCell.value = 1

' We can then take that value and work with it
myValue      = myCell.value
' (explain how this is done)
myValue      = myValue + 1
' Get the next Cell along (B1)
Set myCell   = mySheet.Cells(1,2)
```

```
  ' This will make the value of B1 be 2
  myCell.Value = myValue
```

With Statements

With statements are another way of working with Object which can be handy as an addition to using Variables. This is especially so if we want to do several things with the same Object in tight sequence. It is easiest to show this by an example; here we will set the values of several Cells in sequence.

```
  Set myExcel = CreateObject(“Excel.Application”)
  myExcel.Workbooks.Add()
  Set myWorkbook = myExcel.Workbooks(1)

  ' Here is the with, note the extra indentation
  ' which is here to make the code easier for
  ' humans to read:
  With myWorkbook.Sheets(1)
    .Cells(1,1).value="Row=1,Col=1"
    .Cells(2,1).value="Row=2,Col=1"
    .Cells(1,2).value="Row=1,Col=2"
  End With
```

Key Concept:
Note how anything inside the With (between the With and End With) which begins . is treated as a Method or property of the Object to which the With statement refers.

'Take Home' Ideas from Lesson 2

Objects are stored in Collections to make them easier to find.

In Excel, many Objects have a Collection of Objects as one of their Properties. This allows us to 'drill down' from a top level Object like Excel.Application all the way down to Cells.

Variables are named containers for Objects or non Object data like Strings, Booleans, Dates and Numbers.

Variables are a handy way of being able to find an Object once but use it several times.

Another handy way of using the same Object several times is to use a With statement.

Lesson 3: Loops

Finally we start to make things easier:

I am not you, but if I try to imagine being you, right now I would be thinking:

"This is all very well, but I have just taken an age to do something I could have do in 30 seconds with Excel using just formulae or fills."

That sentiment is perfectly correct! So, it is time to start doing some stuff which is easier and faster to do with scripts than it is by hand, and that usually involves loops.

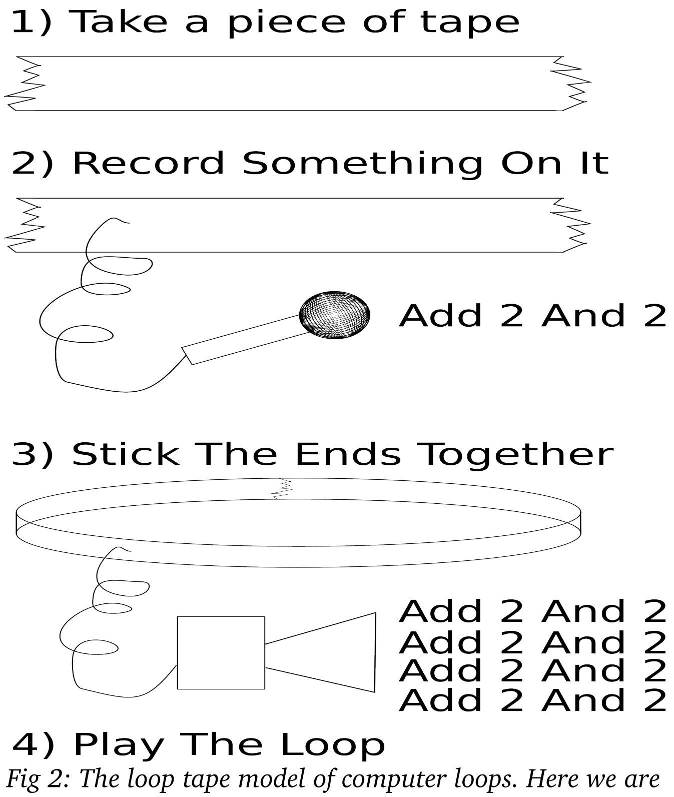


Fig 2: The loop tape model of computer loops. Here we are making a loop which says 'Add 2 And 2' over and over again, for ever!

People do not normally like doing the same thing over and over again; computers love it. Loops are a way of telling a computer to 'loop over' the same set of actions repeatedly. You can think of loops as 'loop tapes'. Figure 2 shows graphically how this mental model works.

The obvious flaw in this approach is that the loop will run for ever. This is called an 'Infinite Loop'. Sometimes in programming this is exactly what you want. Most of the time, however, we want to loop only until some criterion has been met.

For Example:

1. Work down Column A one Cell at a time.
2. Add the contents of the Cell you are on to to its row number.
3. Put the result into the same row in Column B.
4. Stop when you get to an empty Cell in Column A.

That example is exactly the sort of thing one does all the time when Scripting Macros. The way that it automatically stops when a condition is met (when an empty Cell is reached) hints at the power of this technique when compared to just using formulae in Excel.

Drag And Drop Scripting, An Introduction

To make meaning-full examples of how loops can automate repetitive tasks, we need to be able to load pre-existing Excel Workbooks. Again, from experience in the field, the best way to achieve this is via 'drag and drop scripting'.

This approach works by clicking on a Workbook file and dragging it over the icon for our script. When we 'drop' (release the mouse button) Windows runs the script and tells the script the file name of the Workbook you dropped on it. Windows 'tells' you the file name by a special Collection that you can always use from inside the script. This is the *WScript.Arguments* Collection. All scripts of type '.vbs' (VBScript) have a special Object pre-created that is called *WScript*. You can do lots of handy things with *WScript*; reading the names of files which have been dropped onto a script is just one of them.

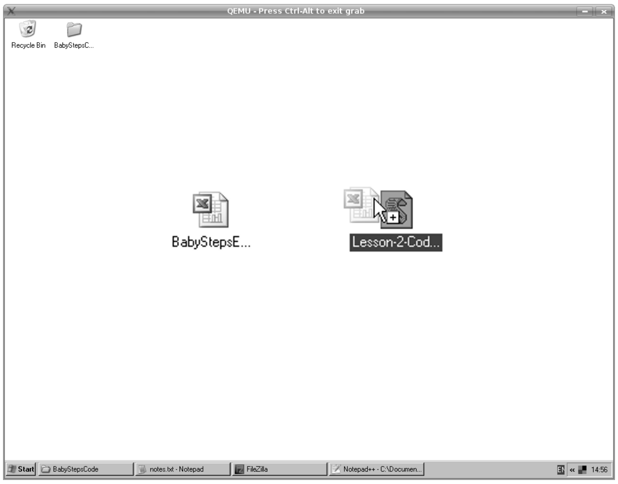


Figure 4: Here is the Workbook file BabyStepsExamples.xls *being 'drag and dropped' onto a Macro Script file. When the script runs, the file name of the Workbook will be be placed in the* WScript.Arguments *Collection of the script at index 0.*

Now that we have figured out how to tell a script which Excel Workbook to open, we need to actually open it. This piece of code will open an Excel Workbook which is dropped onto it:

```
' Create an Excel application Object and make it
' become visible
Set myExcel = CreateObject("Excel.Application")
myExcel.visible=true

' WScript.Arguments is a special Collection into
' which Windows places a list of the file names of any
' files that were dropped on to the script.
' It is a bit unusual because it starts from index 0,
' rather index 1 like most Collections do.
' You just have to remember it as a special case!
myFileName = Wscript.Arguments(0)

' The Workbooks Collection has a special method
' which loads and Workbook file into a new Workbook.
Set myWorkbook=myExcel.Workbooks.Open(myFileName)
```

Now that we have automated the loading of a Workbook we can automate its processing using a script. As we have seen, each Cell Object has a Property *Value*. There is a special Method built into VBScript which can tell if a value is empty. This Method is called

IsEmpty(). It produces a Boolean value of *true* or *false*. It is this handy method that we used to control when our loop will exit. Here is an example script, which I will discuss in more detail afterwards:

```
Set myExcel = CreateObject("Excel.Application")
myFileName = WScript.Arguments(0)
Set myWorkbook=myExcel.Workbooks.Open(myFileName)

' Note that the Sheets Collection starts at
' the more normal index of 1.
Set mySheet=myWorkbook.Sheets(1)

' row is a Variable in which we store the number
' of the current row that the loop is working on.
' We are going to start with row = 2 because
' Row 1 holds the headers. Please also note
' I have called this Variable row just to make
' the script easy to read for humans. It could
' be called anything. For example, I could have
' called the Variable bob as in: bob = 2
row = 2

' This loop will carry on until the current
' Row has an empty Cell in Column 1.
' For a full descripting of this
' loop, see the main text of "Baby Steps"
While Not IsEmpty(mySheet.Cells(row,1).Value)
  mySheet.Cells(row,2).value = _
    mySheet.Cells(row,1).value * 2.0
  row = row  + 1
Wend
```

Note: if you end a line _ it means that the line is continued on the next line. i.e. :

```
a = _
b
```

Is the same as:

```
a = b
```

To test this script we should create an Excel Workbook in which the first Sheet has values something like this:

Price £	Price $
1.5	
2.3	
0.5	

Once the script has run the first Sheet will look like this:

Price £	Price $
1.5	3.0
2.3	4.6
0.5	1.0

To achieve this the script has performed the following steps *(remember that Cells are referenced y,x* i.e. *row,column)*:

1. Started with the Variable *row* with value 2.
2. Checked that Value of Cell(2,1) and found it to be not empty.
3. Taken the Value of Cell(2,1) and multiplied it by 2.
4. Placed the result of the calculation in Cell(2,2).
5. Added 1 to the value of row and put the result back into row.
6. Checked that Value of Cell(3,1) and found it to be not empty.
7. Taken the Value of Cell(3,1) and multiplied it by 2.
8. Placed the result of the calculation in Cell(3,2).
9. Checked that Value of Cell(4,1) and found it to be empty.
10. Not processed the loop any further (sometimes called 'exiting the loop').

For Loops

So far we have looked at *While* loops. These are the most obvious form of loop; they say "keep doing something while something is true." They are not the only form of loop. Another sort that is quite useful is the *For* loop. They say "do something *for* a certain count".

The following two loops are completely identical.

```
' While loop:
bob = 2
While bob < 10
  mySheet.Cells(bob,1)= "Row number = " & bob
  bob = bob + 1
Wend

' For loop:
For bob = 2 To 9
  mySheet.Cells(bob,1)= "Row number = " & bob
Next
```

> *Note: if* `bob = 3` *then* `"Row number = " & bob` *produces* `"Row number = 3"` *i.e. the symbol* `&` *joins a String to a String, Number, Boolean or Date.*

Because, in Excel scripting, we are more often than not looping over data until some data related condition is met, *For* loops are less common than *While* loops. However, as you can see from the above example, when you can use them they take up less space and it is more obvious what is happening.

For Each Loops

We have just seen how For loops are an easy way of performing a set of actions for a set number of repetitions. How about if we want to perform an action on each of the contents of a Collection? In other words *For Each* element in the Collection. Happily, VBScript provides us with a really simple way of doing this.

We could make this happen using a For loop:

```
For index = 1 To myWorkbook.Sheets.Count
  mySheet = myWorkbook.Sheets(index)
  mySheet.Cells(1,1).Value = "Hello World"
Next
```

However, using a For Each is much simpler (and actually faster for the computer to run)!

```
For Each mySheet In myWorkbook.Sheets
  mySheet.Cells(1,1).vValue = "Hello World"
Next
```

'Take Home' Ideas from Lesson 3

There are three types of loops which are really handy: While, For and For Each.

The easiest way of setting up a Scripting Macro to open pre-existing Workbooks is to use Drag-And-Drop-Scripting.

WScript is a pre-existing Object which is available from all scripts. It alows us, amongst many other things, to get the file name of a Workbook which has been 'dropped' onto our script.

A lot of the time we will want to process Cells until we find on or more empty ones. This can be achieved using the handy IsEmpty() method in conjunction with a While loop.

Lesson 4: Getting Tooled Up.

A Quick Note

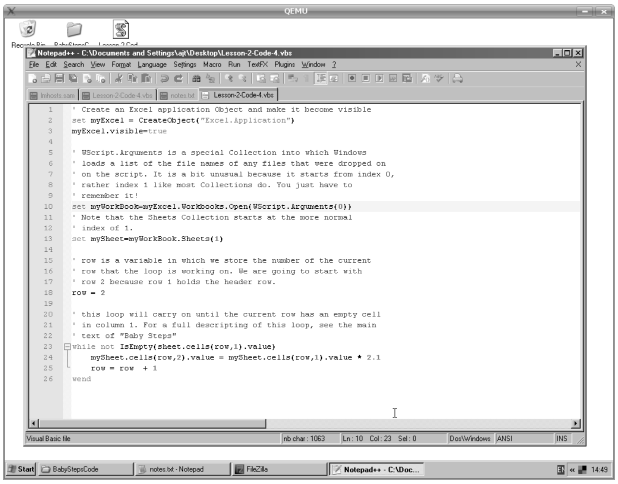
From here onward in Baby Steps, there will be times when it is impossible to fit a whole line of scripting code into a line on the page. To help prevent this being confusing, empty space at the start of script lines will be filled with this special character “░”. If you do no see any of this grey stuff at the start of a line, then the line is a continuation from the line above.

Get A Good Text Editor

As we have seen so far, Macro Scripts are just text files. Text files are unlike documents from things like MS Word because they contain just characters; even new-lines are stored just a characters. We can edit text files with *Notepad* (like we have till now) but it is not a very nice tool for doing this. Before our scripts get much more complex, it will be much easier for us to use a more powerful tool. Fortunately, there are many free text editors in existence. They are so important to the act of programming, that many people have created them.

My favourite text editor for Windows is *Notepad++*. Its name gives the game away. It is a lot like *Notepad* but with loads of extra features like line numbering, text colouring and excellent find/replace tools.

Notepad++ is 'Open Source' and free of charge. This means that you do not have to pay for it at all and you can use it in any way you want other than resell it.



```
' Create an Excel application Object and make it become visible
set myExcel = CreateObject("Excel.Application")
myExcel.visible=true

' WScript.Arguments is a special Collection into which Windows
' loads a list of the file names of any files that were dropped on
' on the script. It is a bit unusual because it starts from index 0,
' rather index 1 like most Collections do. You just have to
' remember it!
set myWorkBook=myExcel.Workbooks.Open(WScript.Arguments(0))
' Note that the Sheets Collection starts at the more normal
' index of 1.
set mySheet=myWorkBook.Sheets(1)

' row is a variable in which we store the number of the current
' row that the loop is working on. We are going to start with
' row 2 because row 1 holds the header row.
row = 2

' this loop will carry on until the current row has an empty cell
' in column 1. For a full descripting of this loop, see the main
' text of "Baby Steps"
while not IsEmpty(sheet.cells(row,1).value)
    mySheet.cells(row,2).value = mySheet.cells(row,1).value * 2.1
    row = row  + 1
wend
```

Figure 4: This is a screen shot of Notepad++ being used to edit one of the example files from "Baby Steps".

To download a copy, please go to this URL:

http://www.sourceforge.net/project/showfiles.php?group_id=95717&package_id=102072.

Alternatively you can do to the package home page and follow the links to download the binary executable files:

http://notepad-plus.sourceforge.net/uk/download.php

Recording Macros

It would be a huge task to learn and remember how to get Excel to do everything via Scripts that we can get it to do via the user interface. Wow, that is scary. Fortunately, we do not have to learn and remember all that! Excel has (as you may well already know) a really handy feature called the 'Macro Recorder'.

The Macro Recorder records 'Internal' Macros; those being ones which are embedded into a Workbook. However, Internal Macros and Scripting Macros as very similar. We can record Internal Macros and then easily translate them into Scripting Macros or into pieces which can be added into Scripting Macros.

First, I will briefly describe recording Macros for this purpose. Then, in the next section, I will explain how to translate the Internal Macros into pieces of Scripting Macro.

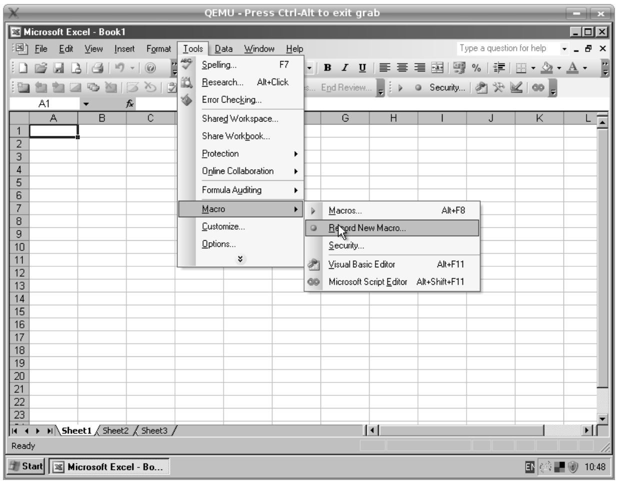
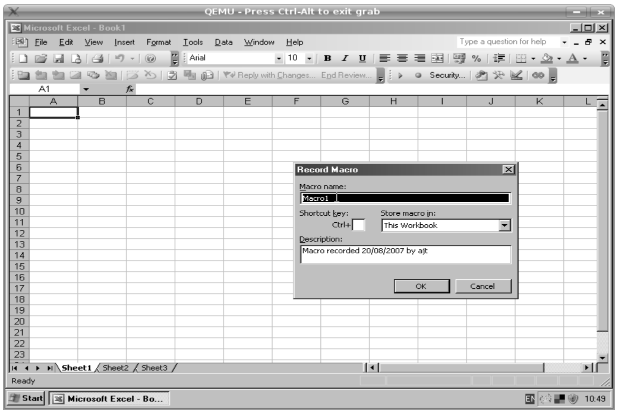


Figure 5: How to start recording a Macro.

The usual use of the Macro Recorder is to create an Internal Macro which will automate a task. Using it requires three simple steps:

1. Start recording.
2. Perform the task.
3. Stop recording.

Starting the recording, in most versions of Excel, is done via the *Tools* menu option. Figure 5 shows this being done and Appendix A has details of how to do it in Excel 2007.


Record Macro

Macro name: Macro1

Shortcut key: Ctrl+

Store macro in: This Workbook

Description: Macro recorded 20/08/2007 by ajt

Figure 6: When choosing a Macro name, you may as well use the one Excel suggests because you are not going to keep the Macro, it is going to be translated into your Script.

Once we have requested to start the Macro Recorder, we need to name the Macro. Because we are not going to keep this Macro (just translate it into a Scripting Macro) it is not very important what we call it; in most cases, the name Excel creates automatically is just fine.

Once we have chosen a name and clicked 'OK' (see figure 6) Excel starts recording. Each action which is taken (e.g. each mouse click or button press) will be recorded as Visual Basic code which will perform the same action. For example, selecting the top Row of Columns *A* to *I* will cause the Macro Recorder to record:

```
  Range(“A1:I1”).Select
```

Unfortunately, we cannot actually see the recording happen as it records. However, it is important to think very carefully as we are performing the tasks which we want recorded; every action will be recorded, including mistakes, 'undos' and accidental Cell selections!

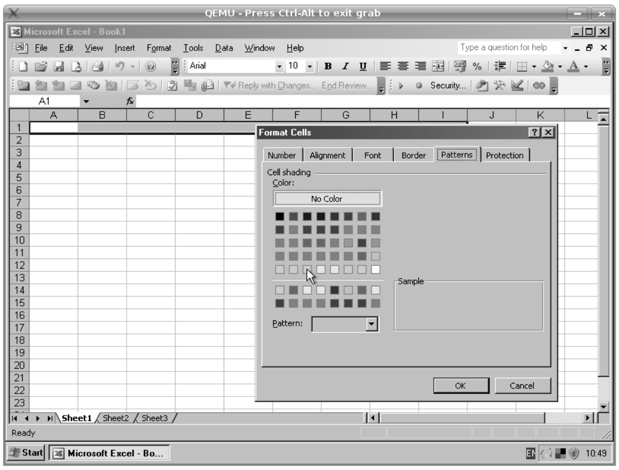


Figure 7: This screen shot shows a set of Cells being coloured whilst the Macro Recorder is running.

Remember that we are not wanting to keep the Macro. We are going to translate it into part of a Macro Script. Because of this, it is much easier to record a separate short Macro for each step of a process than to record a huge Macro of an entire, complex process. In figure 7 we can see a Macro being recorded. This one consists of selecting some Cells and then colouring them. This is about the level of complexity we should be aiming for in any one recording. There are some exceptions where it is not possible to make really short recordings. For example, when we create Pivot Tables in lesson 8.

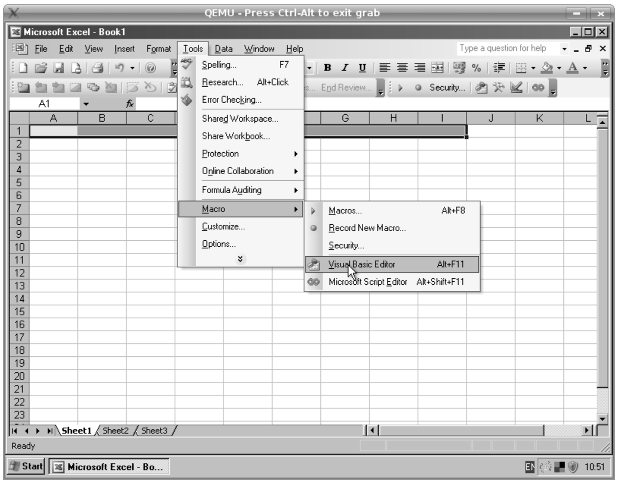
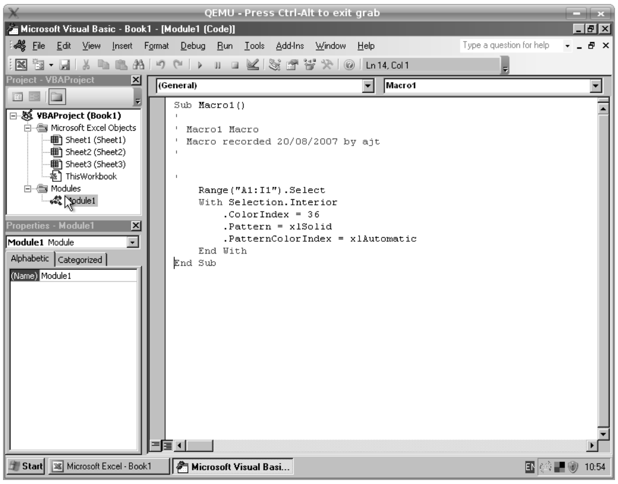


Figure 8: This screen shot shows how, once the actions we want recording have been completed and the Macro Recorder stopped, we can open the Visual Basic Editor and view the Macro code.



```
Sub Macro1()
'
' Macro1 Macro
' Macro recorded 20/08/2007 by ajt
'

'
    Range("A1:I1").Select
    With Selection.Interior
        .ColorIndex = 36
        .Pattern = xlSolid
        .PatternColorIndex = xlAutomatic
    End With
End Sub
```

Figure 9: Here we can see how the code for the Macro has been written into a Module of name 'Module1'. By expanding the Modules node in the left hand tree view and then double clicking on 'Module1' we can see the recorded code.

Translating Recorded Macros

Once we have recorded a Macro, we can use copy and paste to put the Macro code into Nodepad++ (or what ever editor we are using). We are only interested in the code between the line starting “Sub” and the line “End Sub”. Everything else, including these two lines, can be thrown away.

The 'raw' Internal Macro code has to be translated before it will work as a Scripting Macro. Translating macros requires three fundamental steps:

1. The keyboard and mouse based approach of the Macro should be converted to a simpler form for scripting.
2. Excel constants need to be translated into numbers.
3. The results of 1 and 2 need to be integrated into the script so that it operates on the correct Objects.

Keyboard and Mouse to Script Translation

In our example we literally selected some Cells and then changed the background colour of the selection. This is exactly what the Macro Recorder has recorded. A Range Object is identical to a Cell Object (which we have used before) except it refers to one or more Cells at once.

```
  mySheet.Cells(1,1).Value = “Hello World”
```

Is 100% the same as:

```
  mySheet.Range(“A1:A1”).Value = “Hello World”
```

Range and Cell Objects have a Method *Select()* which simply selects them just as though you had selected them on a Worksheet using the mouse. So the Macro Recorder has recorded the Cells being selected and converted this action into the code which calls the *Select()* Method on a Range.

```
  mySheet.Range(“A1:I1”).Select
```

Once the Cells have been selected, the Recorder records the changes to that selection. We should remove this use of a selection and work with the Cells directly:

```
  Range(“A1:I1”).Select
  With Selection.Interior
```

Becomes:

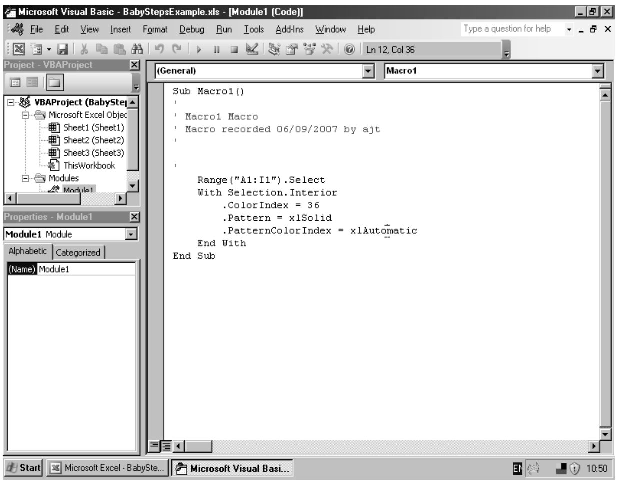
```
    With mySheet.Range(“A1:I1”).Interior
```

The *Interior* Property of a Cell or Range Object is a special Object which Excel uses for controlling the colour, border and other visual Cell features.

Translating Excel Constants

Excel constants are numbers which have special meanings to Excel. For example the number 1 tells the Cell.Interior Object that a Cell border should be solid. The Macro Recorder has special names for these constants. For example instead of using 1 to signify a solid border, it will use *xlSolid*; Excel 'knows' that *xlSolid* actually means 1. Unfortunately, a Scripting Macro does not 'know' what *xlSolid* means so we to tell it.

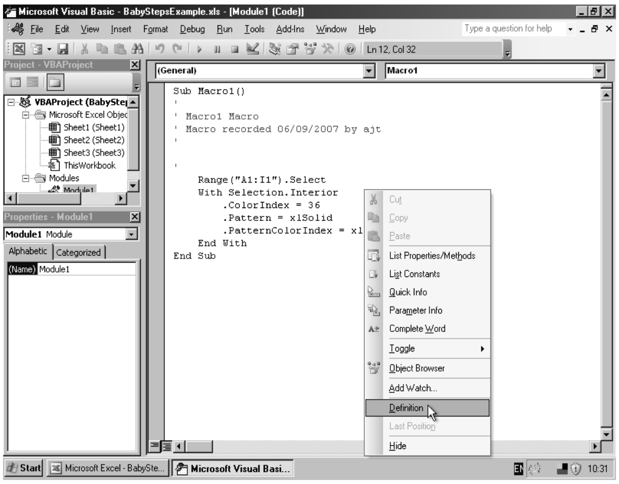
Before we can write the code into a Scripting Macro to define *xlSolid*, or any Excel constant, we need to find out what its value is. Whilst we might be able to remember the value of one or two constants, Excel has hundreds of them. Fortunately for us, Excel also has an easy to use tool for finding out their values.



```
Sub Macro1()
'
' Macro1 Macro
' Macro recorded 06/09/2007 by ajt
'

'
    Range("A1:I1").Select
    With Selection.Interior
        .ColorIndex = 36
        .Pattern = xlSolid
        .PatternColorIndex = xlAutomatic
    End With
End Sub
```

Figure 10: To start the process of finding the value of an Excel constant right click on the constant in the Visual Basic view of the Macro.

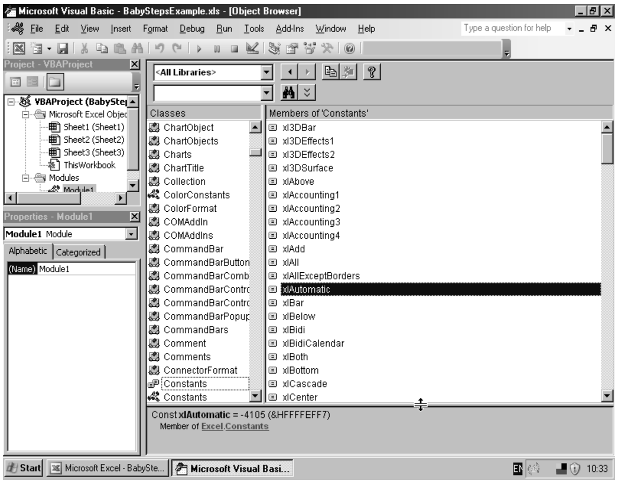
Excel has a tool called the *Object Browser*. We are going to look into using this in more detail in the next section. Using it for finding the value of Excel constants is very simple indeed. That is exactly what we will do now.



```
Sub Macro1()
'
' Macro1 Macro
' Macro recorded 06/09/2007 by ajt
'

'
    Range("A1:I1").Select
    With Selection.Interior
        .ColorIndex = 36
        .Pattern = xlSolid
        .PatternColorIndex = xl
    End With
End Sub
```

Figure 11: The second step to finding out the value of an Excel constant is to click on 'Definition' when context menu comes up.

All that is required is to view the code of our recorded Macro. From this view, we can right click on the text of the constant its self. In our example, if we right click on the text '*xlAutomatic*' a context menu appears. All we need to do is click on '*Definition*'.



Const **xlAutomatic** = -4105 (&HFFFFEFF7)
Member of Excel.Constants

Microsoft Visual Basic - BabyStepsExample.xls - [Object Browser]
<All Libraries>
Classes
Members of 'Constants'
xlAutomatic

Figure 12: Once we have chosen 'Definition' *from the context menu over an Excel constant, the Object Browser will open. In the bottom pane of the Object Browser, we can see the definition of the Excel Constant. In this case,* xlAutomatic *is defined as -4105.*

Once we have clicked on '*Definition*' Excel will automatically start up the Object Browser showing details of the constant on which we clicked. Constants are shown as the number in our normal decimal number system and then in brackets using the more computer centric hexadecimal system. We only need to consider the decimal version.

For our Scripting Macro to use the constants we have to define them in the Scripting Macro. Here is an example of the finished Script segment (this is not a full script):

```
  xlAutomatic = -4105
  xlSolid = 1
  With mySheet.Range("A1:I1").Interior
    .ColorIndex = 36
    .Pattern = xlSolid
    .PatternColorIndex = xlAutomatic
  End With
```

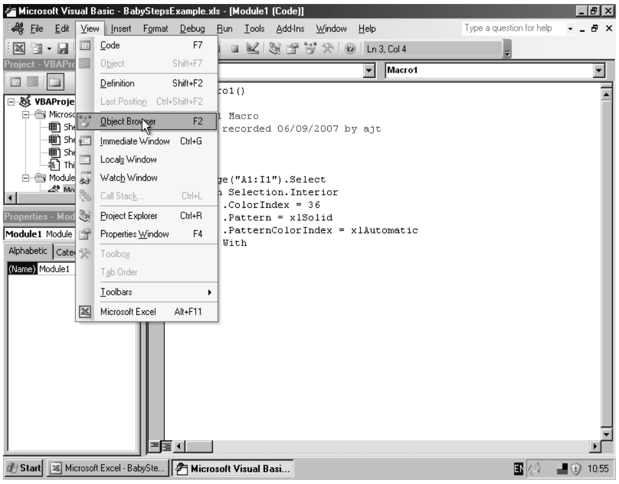
Using The Object Browser

In the previous section we saw how Excel will automatically launch the Object Browser if we ask it for the definition of an Excel constant. We can also manually open the use the Object Browser. This can be very handy to help work out how to do something with Excel.

The Macro Recorder can be very helpful, however sometimes we want to work out how to do something which is hard or fiddly to do with the Macro Recorder. This is where the Object Browser comes in extremely useful.

Just about everything Excel can do from the graphical interface it can also do from the programming interface. This means that our Macro Scripts can do just about anything we want. To allow Excel to have such excellent programmatic control, there are a huge number of Objects and Constants in Excel. These are all known about by the Object Browser. The Object Browser takes its internal description of all these Objects and Constants and displays them in a human usable form. It is not only a browser, but also a search engine. You can use it to search for Objects or Constants by name.

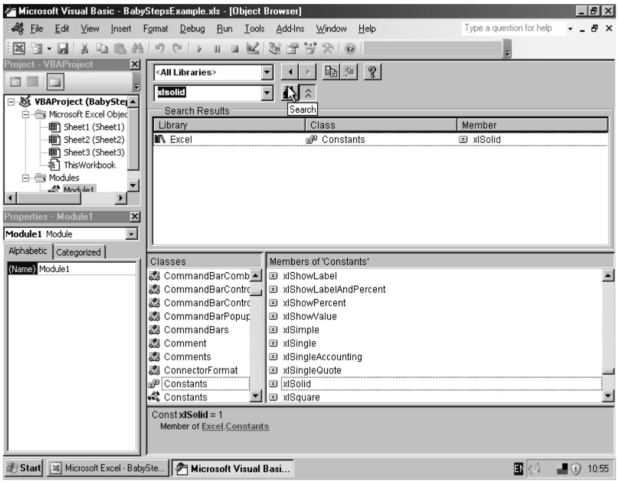
To get the Object Browser started, we first need to be in Visual Basic For Applications view (see figure 13). From there, we can directly open the Browser from the '*View*' menu.



```
ro1()

1 Macro
 recorded 06/09/2007 by ajt



ge("A1:I1").Select
h Selection.Interior
 .ColorIndex = 36
 .Pattern = xlSolid
 .PatternColorIndex = xlAutomatic
With
```

Figure 13: We can start the Object Browser manually from the View menu.

Once we have started the Browser we can start looking around inside the programmatic interface to Excel and so learn a little more about just what we can do with Scripting Macros.



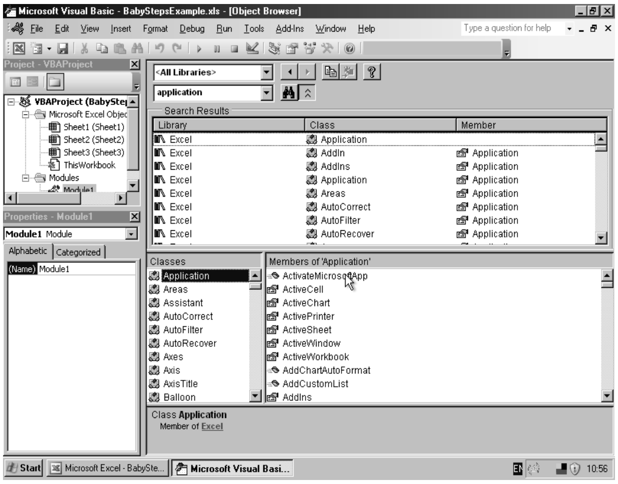
Search Results

Library	Class	Member
Excel	Constants	xlSolid

Classes	Members of 'Constants'
CommandBarComb	xlShowLabel
CommandBarContro	xlShowLabelAndPercent
CommandBarContro	xlShowPercent
CommandBarPopup	xlShowValue
CommandBars	xlSimple
Comment	xlSingle
Comments	xlSingleAccounting
ConnectorFormat	xlSingleQuote
Constants	xlSolid
Constants	xlSquare

Const **xlSolid** = 1
Member of Excel.Constants

Figure 14: This screen shot shows what the Object Browser looks like when it has been opened. Also here I have typed 'xlsolid' *(note that the search is not case sensitive) into the search box and clicked the search button (it has binoculars on it). In the pane at the bottom of the Bowser we can see the result of my search. In the middle panes we can see that* xlSolid *is a member of* Constants.



Library	Class	Member
Excel	Application	
Excel	AddIn	Application
Excel	AddIns	Application
Excel	Application	Application
Excel	Areas	Application
Excel	AutoCorrect	Application
Excel	AutoFilter	Application
Excel	AutoRecover	Application

Classes	Members of 'Application'
Application	ActivateMicrosoftApp
Areas	ActiveCell
Assistant	ActiveChart
AutoCorrect	ActivePrinter
AutoFilter	ActiveSheet
AutoRecover	ActiveWindow
Axes	ActiveWorkbook
Axis	AddChartAutoFormat
AxisTitle	AddCustomList
Balloon	AddIns

Class **Application**
Member of Excel

Figure 15: Not only can we search using the Object Browser – we can browse with it. The bottom two panes allow us to move around the Classes and Collections of Excel and see what is available.

Finding And Removing Bugs

Bugs are a fact of life in anything computer based. Basically, a bug is when our script does not do what we want or expect. There is no simple cure or remedy for bugs. When we see the effect of a bug in the way a script is running we must 'find the bug'. This is just like fixing any machine; the first step is to find out why it is not working properly. Fixing the problem always comes second.

The best advice for finding anything is to develop the best possible way of looking for it. If we are looking for a needle on a carpet, we might use a magnet. If we are looking for a star, we might use a telescope. So, we need tools to help us find bugs in Scripting Macros. Fortunately for us there are three:

1) Our brains! We must not make any assumptions. The human brain is amazing powerful at the art of figuring out what is going on, as long as we do not assume we already know what is going on. Finding bugs requires an open mind and lots of concentration.

2) Wscript.Echo: This Method pops up alert boxes that let us see the inner workings of a script. One way of looking for a bug is to imagine all the values of Variables as the script runs. An alternative is to imagine which pieces of code will be running at which point in time. Placing Wscript.Echo statements in

the scripts allows us to 'see' if our imaginings are correct. If they are not, maybe that points to the bug. For example, what values will *myNumber* take on?

```
  For row = 1 To 10
    myNumber = (row * row) / (row * 0.25)
  Next
```

To find out, simply just get the script to tell us:

```
  For row = 1 To 10
    myNumber = (row * row) / (row * 0.25)
    Wscript.Echo myNumber
  Next
```

If having ten popup boxes is just too annoying then this is an alternative:

```
  myList = “”
  for row = 1 to 10
    myNumber = (row * row) / (row * 0.25)
    myList = myList & myNumber & “,”
  next
  Wscript.Echo myList
```

3) Option Explicit: This s a simple directive that we can put at the top of scripts. It forces the VBScript interpretor (the thing that actually runs our Scripting Macro) to check that you have explicitly use Dim to create all Variables before you use them.

Here is an example of the use of Option Explicit to help debug a script:

In this example, the script will produce meaning less results:

```
  ' Create Excel Etc
  ...
  ' Populate a Spreadsheet
  for row=1 to 100
    value=row * 10
    mySheet.Cells(row,1).Value=valeu
  next i
```

A mistake like this is a large script can take a very long time to spot using just the human eye. The solution is to do this:

```
  Option Explicit
  Dim mySheet,row,value
  ' Create Excel Etc
```

```
...
' Populate a Spreadsheet
for row=1 to 100
  value=row * 10
  mySheet.Cells(row,1).Value=valeu
next i
```

The 'Option Explicit' at the top of the script tells VBScript not to use a Variable unless it has been created with a Dim statement. In this case, we have not created a Variable *valeu* so the script will not run; instead it will give us a warning telling us that the Variable is not defined and the line on which the bug is.

'Take Home' Ideas from Lesson 4

From now on scripts will have “░” instead of spaces at the start of new lines.

Scripts are text and so are best edited with a text editor like notepad++.

Excel's Internal Macros and Scripting Macros are very similar. A good approach to figuring out how to do something with a script is to record an Excel Internal Macro and then translate it into a Scripting Macro.

Script translation requires 3 steps:

1) *Covert from the keyboard and mouse based approach to a simpler direct methodology.*
2) *Translate constants.*
3) *Integrate.*

To translate constants, we can right click on them in Excel's code view and go to “Definition”.

Finding definitions uses the Object Browser. The Object Browser is very useful for looking at other parts of Excel's programmatic interface.

Getting rid of bugs takes 3 techniques: Keeping and open mind, using Wscript.Echo to 'see' what is really happening and always using Option Explicit.

Lesson 5: Starting To Work With Data

Special Folders

In this lesson we are going to do some real data processing. To do this we need to find Workbooks to open and find good locations to save new Workbooks. To do this we can use Special Folders. So, please be patient; this section will only be brief, then we can start with some real work!

Special Folders are folders that are or at least potentially can be present on all Windows computers; these include the *My Documents*, *Fonts* and *Start Menu* folders. There are two types of Special Folders: Those that map to standard directories and those that do not. The *Favourites* folder, for example, maps to a standard directory; the *My Computer* folder does not. We are only interested in Special Folders which map to (represent) real folders.

The WScript.Shell SpecialFolders collection contains the full path to each of the special folders that we can access using Scripting Macros. The table below lists the Keys to the Collection and the contents of each of the special folders in the SpecialFolders collection.

Identifier	Folder Contents
AllUsersDesktop	Shortcuts that appear on the desktop for all users
AllUsersStartMenu	Shortcuts that appear on the Start menu for all users
AllUsersPrograms	Shortcuts that appear on the Programs menu for all users
AllUsersStartup	Shortcuts to programs that are run on startup for all users
Desktop	Shortcuts that appear on the desktop for the current user
Favorites	Shortcuts saved as favorites by the current user
Fonts	Fonts installed on the system
MyDocuments	Current users documents
NetHood	Objects that appear in Network Neighborhood
PrintHood	Printer links
Recent	Shortcuts to current users recently opened documents

Identifier	Folder Contents
SendTo	Shortcuts to applications that show up as possible send-to targets when a user right-clicks on a file in Windows Explorer
StartMenu	Shortcuts that appear in the current users start menu
Startup	Shortcuts to applications that run automatically when the current user logs on to the system
Templates	Application template files specific to the current user

The way to use the Special Folder feature of a WScript.Shell object is via the *.SpeciaFolders.Item()* Method. The script below finds the location of the MyDocuments special folder and displays it to us.

```
Option Explicit
Dim myShell, myPath
Set myShell = WScript.CreateObject("WScript.Shell")
myPath = myShell.SpecialFolders.Item("MyDocuments")
WScript.Echo "MyDocuments='" & myPath & "'"
```

Using 'SendTo' As A Really Easy Way To Run Scripts

The script below creates a shortcut to a Scripting Macro 'myMacro.vbs' in the SendTo special folder which means we can then right click on any Workbook file and send it to that Scripting Macro. The script assumes that our Scripting Macro is in MyDocuments\ScriptingMacros.

```
Option Explicit
Dim  myShell, sendToFolder, myScriptPath, myShortcut

Set myShell = WScript.CreateObject("WScript.Shell")
sendToFolder = myShell.SpecialFolders("SendTo")
myScriptPath = myShell.SpecialFolders("MyDocuments") & 
"\myMacro.lnk"

Set myShortcut = myShell.CreateShortcut(sendToFolder & 
"myMacro.lnk")
myShortcut.TargetPath =  myScriptPath
myShortcut.Save
```

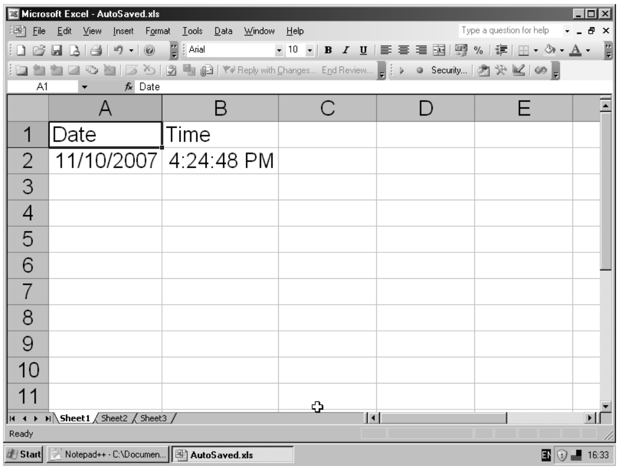
Saving And Opening From Special Folders

When we use Scripting Macros to automate a task we want the input and output files to be placed in easy to find places. Microsoft has carefully moved us into thinking in terms of MyDocuments and Desktop *etc*.; this makes special folders like this ideal for automated scripting tasks.

Below are two examples. The first script creates a new Workbook. It then places the current time and date into one of the Sheets. Once it has done this it saves the Workbook onto the Desktop. It automatically overwrites any existing Workbook of the same name. You can tell that it has done this because each time the script runs the date and/or time will be updated.

The second script is even simpler; it just opens the Workbook the first script created. The script requires no user intervention because it can find the Workbook using the special folder 'Desktop'.

```
  ' This script saves a Workbook to the Desktop
  Option Explicit
  Dim myShell, myExcel, myWorkbook, mySheet, myShell,
xlNormal

  ' Open Excel
  Set myExcel = CreateObject("Excel.Application")
  ' Create a Workbook and set myWorkbook to refer to it
  ' all in one go
  Set myWorkbook = myExcel.Workbooks.Add
  Set mySheet = myWorkbook.Sheets(1)
  mySheet.Cells(1,1).Value="Date"
  mySheet.Cells(2,1).Value=Date()
  mySheet.Cells(1,2).Value="Time"
  With mySheet.Cells(2,2)
    .Value=Now()
    .NumberFormat = "[$-409]h:mm:ss AM/PM;@"
  End With
  mySheet.Columns(1).AutoFit

  ' Tell Excel not to complain about overwrite files and
  ' other things like that
  myExcel.DisplayAlerts = FALSE

  ' Save Excel Workbook to Desktop
  xlNormal = -4143
  Set myShell=CreateObject("WScript.Shell")
  myWorkbook.SaveAs myShell.SpecialFolders("Desktop") &
"\AutoSaved.xls",xlNormal
  myExcel.Quit
```


Microsoft Excel - AutoSaved.xls
File Edit View Insert Format Tools Data Window Help
Type a question for help
Arial
10
Reply with Changes... End Review...
Security...
A1
Date
Sheet1
Sheet2
Sheet3
Ready
Start
Notepad++ - C:\Documen...
AutoSaved.xls
16:33

	A	B	C	D	E
1	Date	Time			
2	11/10/2007	4:24:48 PM			
3					
4					
5					
6					
7					
8					
9					
10					
11					

Figure 16: This screen shot is result of the first of the two Desktop scripts. This is the Excel Workbook that I opened from double clicking on "AutoSaved.xls" on the desktop.

```
 ' This script opens the Workbook saved by the
 ' previous one
Dim myShell, myExcel,myShell
Set myExcel = CreateObject("Excel.Application")
Set myShell=CreateObject("WScript.Shell")
myExcel.Open myShell.SpecialFolders("Desktop") &
"\AutoSaved.xls"
```

Merging Between Workbooks

In my forward I talked about chains of Excel spreadsheets. These are caused where data passes through many steps and/or many hands. One key action in such chains is data merging. Often we must take two or three data sets and merge them together.

Moving Columns of data around by hand can be tedious. Searching through rows of data to find matches can be soul destroying. Scripting Macros can come to the rescue.

Merging Data By Example

To set the scene of data merging, a good example will do nicely. By looking at a plausible business scenario we can truly imagine the steps involved in merging.

A fairly common cause for the need to merge is 'incomplete data'. Consider that we are in a sales department. Sales people are out "on the road" collecting orders. They report the orders back to head office on spreadsheets like this:

Customer Name	Part Number	Quantity	Date
ABC Engineering	12345	1	11/11/2007
XYZ Fabrication	7234	4	6/11/2007

Our job is to record there orders and get them dispatched to fulfilment from where the parts will be sent to the customers.

But notice, the order records from the sales people do not tell us enough for fulfilment to do their job. All we have is a customer name. To overcome this we have another "master customer list". This is the critical core Workbook from which all customer information is run. In this Workbook is a Sheet with customer contact details like this:

Name	Id	Address	Tel	Fax	Main Contact
ABC Engineering	5	45 False Road	012345678	012345679	John
XYZ Fabrication	8	23 Nowhere Avenue	087564321	012345672	Jane

Our first challenge is to write a Scripting Macro which will output a Sheet to send to fulfilment. This script will take each order Row from a sales person and match the customer name against the *Name* column in the *Customer Contact Details* Sheet. The output of this process should have the customer name, address, part number, quantity and order date.

Dictionary Objects

Dictionary Objects are very useful things indeed. They let us just "look up" values rather than having to go and search for them. In the case of our example, we can store all the customer addresses and 'look them up' by customer name. The thing we are looking up is a "value" and we look it up by a "key". Here is a description of all the Properties and Methods of a Dictionary Object.

Dictionary Object Properties:

CompareMode	This sets the compare mode for using different character sets. We can normally

	ignore this.
Count	The number of key/value pairs in this Dictionary.
Item(key)	Return the value looked up by key. We can also set the value for an existing key: myDictionary.item(“fred-wife”)= “wimla”
Key(key)	Allows us to change a key. Consider that Wilma remarried Barny then we could put: myDictionary.key(“fred-wife”)=“barny-wife”

Dictionary Object Methods:

Add key,value	Places a new key/value pair into the Dictionary.
Exists(key)	Returns TRUE is key is in the Dictionary and FALSE otherwise.
Remove(key)	Removes a key/value pair from the Dictionary.
Items()	Returns an array of all the values in the Dictionary.
Keys()	Returns an array of all the keys in the in the Dictionary.
RemoveAll()	Totally empties the Dictionary so it has no key/value pairs at all.

So, to merge orders with addresses we can store all the customer names as keys. For each customer name key we store the address as the value.

Let us use a customer called “hack and slash” with an address “123 Fake Street”. we can then store the address with the names as a key like this:

```
Dim myDictionary
Set myDictionary=CreateObject(“Scripting.Dictionary”)
myDictionary.Add “hack and slash”, “123 Fake Street”
```

The reverse of this is to look up the value associated with an already stored key. We can demonstrate this with a continuation of the previous script.

```
Dim address
address = myDictionary.Item(“hack and slash”)
Wscript.Echo address
```

The above script will echo “123 Fake Street”

Simple Merge Script

To make a merge script work it will have to:

1) Load the order sheet.
2) Load the customer information sheet.
3) Create a new sheet.
4) Populate the new sheet with a merged data set.
5) Save the new sheet.

To make this nice and simple I will say that the customer information Workbook is already on my Desktop. This means that we can use the 'special folders' we have already looked at to locate the Workbook. To continue with the easy concept, I will put the output Workbook on the Desktop as well. Finally, using 'drag and drop' seems a sensible choice for loading the sales Workbook into my Scripting Macro.

To do the data merge the script will have to look up addresses by customer name. So it will load a Dictionary; the keys will be customer names and the values will be customer addresses. Next the script will work its way down the sales Spreadsheet (from the sales Workbook that was dropped onto it). For each Row on the sales Spreadsheet the script will read in the customers name and look up the address.

Once it has found the address it will write the name, address, order quantity and date into the orders Sheet in the output Workbook. The following script performs this simple merge:

```
Option Explicit
Dim myShell,myExcel,ordersWB
Dim customersWB,outputWB,myDict
Dim desktop,row,ordersSh,customersSh,outputSh
Set myExcel=CreateObject("Excel.Application")
Set myDict =CreateObject("Scripting.Dictionary")
Set myShell=CreateObject("WScript.Shell")
desktop=myShell.SpecialFolders("Desktop")

myExcel.Visible=TRUE
```

```
  Set ordersWB=myExcel.Workbooks.Open(desktop &
"\orders.xls")
  Set customersWB=myExcel.Workbooks.Open(desktop &
"\customers.xls")
  Set outputWB=myExcel.Workbooks.Add()

  Set ordersSh=ordersWB.Sheets(1)
  Set customersSh=customersWB.Sheets(1)
  Set outputSh=outputWB.Sheets(1)

  ' This loads the dictionary with the
  ' addresses keyed by name
  row=2
  While(NOT IsEmpty(customersSh.Cells(row,1)))
    myDict.Add
customersSh.Cells(row,1).Value,customersSh.Cells(row,3).
Value
    row=row+1
  Wend

  ' Lay out the output sheet
  outputSh.Cells(1,1).Value="Customer Name"
  outputSh.Cells(1,2).Value="Part Number"
  outputSh.Cells(1,3).Value="Quantity"
  outputSh.Cells(1,4).Value="Date"
  outputSh.Cells(1,5).Value="Address"

  ' This scans the orders and creates the output
  row=2
  While(NOT IsEmpty(ordersSh.Cells(row,1)))
    outputSh.Cells(row,1).Value=ordersSh.Cells(row,1)
    outputSh.Cells(row,2).Value=ordersSh.Cells(row,2)
    outputSh.Cells(row,3).Value=ordersSh.Cells(row,3)
    outputSh.Cells(row,4).Value=ordersSh.Cells(row,4)
    ' This is the merge, looking up the address from
    ' the Dictionary
    outputSh.Cells(row,5).Value= _
      myDict.Item(ordersSh.Cells(row,1).Value)
    row=row+1
  Wend

  'Clean up output
  outputSh.Columns("D:D").NumberFormat = _
    "[$-F800]dddd, mmmm dd, yyyy"
  outputSh.Columns("A:E").AutoFit
  outputSh.Rows("1:1").Font.FontStyle = "Bold"
```

This script produces this output when one of the orders is for “abc engineering”:

Customer Name	Part Number	Quantity	Date	Address
ABC Engineering	12345	1	11-Nov-2007	45 False Road
XYZ Fabrication	7234	4	6-Nov-2007	23 Nowhere Avenue
abc engineering	12348	12	15-Nov-2007	

This script is all well and good but it will be no use in the real world. For example, the customer information Workbook may well have a customer names as "ABC Engineering"; however, a sales person (maybe in a rush) might put "abc engineering" (see the above table). The sales person might even put "abcengineering". How can we deal with these situations?

The solution is to this real world situation is a bit of pragmatism. Step one is to remove all unnecessary information. "ABC Engineering" and "abc engineering" are clearly the same company. We can change the script to ignore the case of names to reflect this reality. This can be done by making all the keys in the Dictionary lower case via the LCase() function which is built into VBScript. The other built in Function to use here is Trim() which removed any unwanted spaces from the start or end of a String.

```
Dim MyString
myString = "VBSCript "
myString = LCase(Trim(myString))
' We can see here that the output is all lower case
' and the trailing space has gone.
Wscript.Echo "'" & myString "'"
```

Our second issue is where "ABC Engineering" is written "abcengineering" or even "bcaengineering". These are examples of genuine mistakes made by the sales person. If we try to engineer some hyper-complex script to work out the answer despite incorrect inputs, we will end up wasting a lot of time.

The only thing really qualified to sort out a real human error is another human. So we will create an extra Sheet of errors we can go through and fix by hand.

Working Merge Script

```
Option Explicit
```

```
  Dim
myShell,myExcel,ordersWB,customersWB,outputWB,myDict
  Dim desktop,row,ordersSh,customersSh,outputSh,key
  Dim errorSh,errorRow,okRow
  Set myExcel=CreateObject("Excel.Application")
  Set myDict =CreateObject("Scripting.Dictionary")
  Set myShell=CreateObject("WScript.Shell")
  desktop=myShell.SpecialFolders("Desktop")

  myExcel.Visible=TRUE
  Set ordersWB=myExcel.Workbooks.Open(desktop &
"\orders.xls")
  Set customersWB=myExcel.Workbooks.Open(desktop &
"\customers.xls")
  Set outputWB=myExcel.Workbooks.Add()

  Set ordersSh=ordersWB.Sheets(1)
  Set customersSh=customersWB.Sheets(1)
  Set outputSh=outputWB.Sheets(1)
  Set errorSh=outputWB.Sheets(2)

  ' This loads the dictionary with the
  ' addresses keyed by name
  row=2
  While(NOT IsEmpty(customersSh.Cells(row,1)))
    myDict.Add
LCase(Trim(customersSh.Cells(row,1).Value)),customersSh.
Cells(row,3).Value
    row=row+1
  Wend

  ' Lay out the output sheet
  outputSh.Cells(1,1).Value="Customer Name"
  outputSh.Cells(1,2).Value="Part Number"
  outputSh.Cells(1,3).Value="Quantity"
  outputSh.Cells(1,4).Value="Date"
  outputSh.Cells(1,5).Value="Address"

  errorSh.Cells(1,1).Value="Customer Name"
  errorSh.Cells(1,2).Value="Part Number"
  errorSh.Cells(1,3).Value="Quantity"
  errorSh.Cells(1,4).Value="Date"
  errorSh.Cells(1,5).Value="Address"

  ' This scans the orders and creates the output
  ' We need three different Row Variables because the
  ' Row we write to is not the same as the number
  ' as the one we read from any more
  row=2
  errorRow=2
  okRow=2
  While(NOT IsEmpty(ordersSh.Cells(row,1)))
    key=Trim(LCase(ordersSh.Cells(row,1).Value))
    ' Here is the decision if the address can be looked
up or not
    If myDict.Exists(key) Then
      ' Yes it can - so put the Row into the ouput Sheet
      outputSh.Cells(okRow,1).Value= _
        ordersSh.Cells(row,1)
      outputSh.Cells(okRow,2).Value= _
        ordersSh.Cells(row,2)
      outputSh.Cells(okRow,3).Value= _
        ordersSh.Cells(row,3)
      outputSh.Cells(okRow,4).Value= _
        ordersSh.Cells(row,4)
      outputSh.Cells(okRow,5).Value= _
        myDict.Item(key)
      okRow=okRow+1
    Else
      ' No it cannot - so put the Row into
      ' the error Sheet
      errorSh.Cells(errorRow,1).Value= _
        ordersSh.Cells(row,1)
      errorSh.Cells(errorRow,2).Value= _
        ordersSh.Cells(row,2)
      errorSh.Cells(errorRow,3).Value= _
        ordersSh.Cells(row,3)
    errorSh.Cells(errorRow,4).Value= _
        ordersSh.Cells(row,4)
      errorRow=errorRow+1
    End If
    row=row+1
  Wend

  'Clean up output
  outputSh.Columns("D:D").NumberFormat = "[$-F800]dddd,
mmmm dd, yyyy"
  outputSh.Columns("A:E").AutoFit
  outputSh.Rows("1:1").Font.FontStyle = "Bold"
  outputSh.Name="Orders To Process"

  errorSh.Columns("D:D").NumberFormat = "[$-F800]dddd,
  mmmm dd, yyyy"
  errorSh.Columns("A:D").AutoFit
  errorSh.Rows("1:1").Font.FontStyle = "Bold"
  errorSh.Name="Errors"
```

Here is a step by step approach to getting data merging to work:

1) Find a Column which is in both Sheets
2) Work out which Sheet has the “master” records and which the “slave”. In our example the customer information Sheet was the master. Do not worry if it is not obvious which is which; you can always try it both ways and see which works better.
3) Find the Column in the master which is shared with the slave (e.g. customer name).
4) Figure out all the variability which can be removed from the shared column data. Examples are capitals, spaces, punctuation.
5) Load the data you want from the master into a Dictionary Object. Key the Dictionary by the share column.
6) For each row in the slave, look up the data in the Dictionary.
7) If a matching entry in the Dictionary is found, put the data from the slave and the Dictionary into the output Sheet.
8) If no match is found, put the data from the slave into the 'errors' Sheet.

Merging Multiple Columns

So far we have looked at merging multiple columns from the slave Sheet with a single column (e.g. address) from the master. How could we merge multiple columns from both Sheets? It would be quite hard to load more than one Column into the Value of a key/value pair in the Dictionary Object so that is probably not the solution.

As it turns out, the solution is very simple. Do everything the same way as for the single master Column merge, but this time store the Row number in the master not the Column data. Each time a match is found in the Dictionary, the Row number can be used to extract the data for the output directly from the master Sheet.

'Take Home' Ideas from Lesson 5

Wscript.Shell has a Collection called SpecialFolders.

The Special Folder Collection is very useful indeed for finding easy to use places on the computer's hard drive like the Desktop.

Dictionary Objects let us store values and then look these values by keys.

Merging data from multiple spreadsheets is a very common and important Scripting Macro task.

Merging can be done quick and easily with the help of Dictionaries.

Lesson 6: Reading & Writing Files

What Is A File?

This might seem an odd question. We all use computer files all the time. We open them, save to them, upload them and download them. However, do we really know what they are? Like many every-day items, we often learn to use them without truly knowing them.

Thankfully files are simple. They are simpler than most people would imagine. A file is just a long list of numbers. Each number can take a value between 0 and 255. That is it!

We may have extra stuff associated with the file; for example, we might give it a name. But the name is not part of the file; the file is just a list of numbers. We might make a place where people or programs can find our file. We often call these places folders or directories. However, a folder is not the file. Neither is the name we use to help locate the file inside the directory; that name is part of the directory not part of the file. A file is just a list of numbers.

All this begs a question “If files are so very simple, how come we can do so many amazing things with them?”

The answer to this lies in a simple child's game. Often children seek to make a easy to use cypher by replacing letters with numbers:

a=1, b=2, c=3, d=4 etc.

To write down “a·b·c” we could put “1·2·3”. Now the numbers have taken on a meaning. They are still just numbers, but we have put special meanings onto them. Using a simple scheme just like this we can store what we think of as text as a list of numbers. Yes, that means we can store text in a computer file! The file does not 'know' that it contains text; it is just a list of numbers. However, we can interpret those numbers as representing characters and so, to us, the file contains text.

We could consider the file which contains the digital version of “Baby Steps”. This has to hold much more than just text; it as images, tables and complex structure information. How, how can such complexity be stored in a simple list of numbers between 0 and 256? This is where format comes into play. Format takes sequences of numbers and give them meaning. This is just like in English writing “w·r·i·t·i·n·g” has more meaning than the sum of it individual letters.

File formats can become very complex indeed. In “Baby Steps” we are not interested in complex so we will keep things simple. It is amazing what can be done simply with files and Excel.

For most of the rest if this book we will only deal with what are called “text files”. These have direct translations of the numbers in the file to characters, punctuation and a few layout rules. Text files are so simple that they can be read and converted to human readable from using Notepad. Notepad (or Notepad++) reads in each number from the text file and converts it into a human readable character *etc*.

When we write text in Notepad, every so often we will press the 'enter key' to drop to the next line. Years ago some computers called the enter key 'new-line'. This is because in text file editors like Notepad, pressing it produces a special layout character which means “start a new line”. This is an example of one of the hand full of special layout characters in text files.

New-line characters are stored in the text file as the number 13. Space is another example with its number being 32. The numeral “1” is represented by the number 48 and the letter “A” by 64. So “123 A” is 48·49·50·32·64.

When we work with text files we tend to let the computer programs we work with do all the translation between the files' numbers and the text we are working with. In other words, we read text files into Strings and we write Strings to text files. Towards the end of the book I discuss what are called binary files. These are where the actual numbers are read and written directly. We almost never need to directly handle binary files in Excel Macro Scripting.

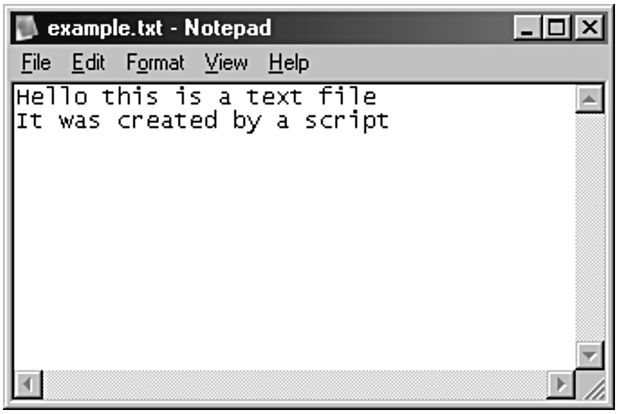
Writing To A Text File

Macro Scripting uses a special Object to access files directly. To write to a file we create a System.FileSystemObject. We give this Object information on where to put the file that we will be writing, its name and if it should replace any pre-existing file with the same name in the same folder. From this information the FileSystemObject creates a TextStream Object. It is this TextStream Object which actually does the writing.

The example below will create a text file on the Desktop ad inside that file it will place two lines of text.

```
Option Explicit
Dim myFSO, myTSO,myShell,myName,forWriting
forWriting=2
```

```
Set myFSO = CreateObject("Scripting.FileSystemObject")
Set myShell=CreateObject("WScript.Shell")
' The file location (folder) and name are sent to the
' FSO as a single String: Folder Name \ File Name
myName=myShell.SpecialFolders("Desktop") &
“\example.txt”
' The arguments to the OpenTextFile method here
' give the location\name, that we want to write
' and that it is OK to create the file if it does
' not already exist
Set myTSO = myFSO.OpenTextFile(myName,forWriting,true)
myTSO.WriteLine("Hello this is a text file")
myTSO.WriteLine("It was created by a script")
' It is good practice to always close a TextStream
myTSO.Close
```



```
Hello this is a text file
It was created by a script
```

Figure 17: This is what Notepad shows if we use it to open the file “example.txt” created by our file file writing script.

The previous script will always replace an existing file with a new one. This means that it 'secretly' performs the following steps:

1) Check if a file of that Folder\Name exists.
2) If it does, delete it permanently.
3) Create a new file with the same Folder\Name.
4) Write to the new file.

What happens if we just want to add to the file rather than replace it? This is not so odd; we might want to write a line a the end of as file every time a script is run to act as an audit trail for example. To do this we tell the FileSystemObject to 'Append' rather than 'Write' when we open the file and create the TextStream Object.

This script adds a new line giving the date and time the script was run to a file “script.log”. The file will be created on the Desktop the first time the script runs. After that, a line will be added to the existing file each time it is run again.

```
Option Explicit
Dim myFSO,  myTSO, myShell, myName, forWriting, 
forAppending
forWriting   = 2
forAppending = 8
Set myFSO = CreateObject("Scripting.FileSystemObject")
Set myShell=CreateObject("WScript.Shell")
' The file location (folder) and name are sent to the
' FSO as a single String: Folder Name \ File Name
myName=myShell.SpecialFolders("Desktop") & 
"\script.log"
' The arguments to the OpenTextFile method here
' give the location\name, that we want to write
' and that it is OK to create the file if it does
' not already exist
Set myTSO = 
myFSO.OpenTextFile(myName,forAppending,true)
myTSO.WriteLine("Script Run: " & now)
myTSO.Close
```

I ran this script a few time and here is the contents of the resulting file:

```
Script Run: 18/10/2007 13:03:31
Script Run: 18/10/2007 13:04:30
Script Run: 18/10/2007 13:04:31
Script Run: 18/10/2007 13:04:42
Script Run: 18/10/2007 13:04:44
Script Run: 18/10/2007 13:04:45
Script Run: 18/10/2007 13:04:46
```

Reading From A Text File

Now that we can write files, how about reading them? First of all, why would we want to read one? Excel can read Excel files so why do we need a different way of reading files?

Sometimes we need to use a script to read a file, work with what it has read and then put the result in Excel. For example, we might have a file that has a formate like this:

```
Begin
User = AJT
```

```
Date = 01/07/2007
Action = logon
End
Begin
User = FF
Date = 01/08/2007
Action = logoff
End
```

In this file we know that a line which says “Begin” starts a block of lines and a line which says “End” ends that block. This extra information about the meaning of the contents of the file is what we call format. This format is not one that Excel understands so we have to load the data into a spreadsheet using a script:

```
  Option Explicit
  Dim
myFSO,myExcel,myWorkbook,mySheet,myTS,row,line,keyWord,d
atum

  ' Another way of defining Variables which we will not
  ' change is to use the “Const” keyword
  Const forReading = 1

  Set myExcel=CreateObject("Excel.Application")
  Set myWorkbook=myExcel.Workbooks.Add()
  Set mySheet=myWorkbook.Sheets(1)
  myExcel.Visible=TRUE
  With mySheet
    .Cells(1,1).Value="User"
    .Cells(1,2).Value="Date"
    .Cells(1,3).Value="Action"
  End With

  ' Set row to the first Row we
  ' want to write to in Excel
  row=2

  ' Open our file in a TextStream Object
  Set myFSO=CreateObject("Scripting.FileSystemObject")

  ' This is how we open a file reading using the
  ' FileSystemObject
  Set myTS=myFSO.OpenTextFile _
    ( _
      "myFile.txt", _
      forReading, _
      FALSE _
    )
```

```
' This means keep looping until the end of the file
While NOT myTS.AtEndOfStream
    ' Read a line of text and put it in
    ' the Variable line
    line=myTS.ReadLine()
    ' Split the line in two iff it has a = in it
    line=Split(line,"=")
    'See if there are two parts or one
    if UBound(line)=1 Then
        datum=Trim(line(1))
    end if
    ' It will always have at least one part once split
    ' Strip spaces off the beginning and end
    ' at the same time
    keyWord=Trim(line(0))
    'Make sure the key word is all lower case
    keyWord=LCase(keyWord)
    ' Choose what to do depending on the line
    if keyWord="end" Then
        ' At the end of a block, move down
        ' one row in Excel
        row=row+1
    elseif keyWord="user" Then
        mySheet.Cells(row,1).Value=datum
    elseif keyWord="date" Then
        mySheet.Cells(row,2).Value=datum
    elseif keyWord="action" Then
        mySheet.Cells(row,3).Value=datum
    end if
Wend
' For neatness - close the TextStream
myTS.Close
' Tidy up our spreadsheet
mySheet.Columns(1).Autofit
mySheet.Columns(2).Autofit
mySheet.Columns(2).Autofit
```

Reading And Writing In Chunks

Most of the time reading and writing lines is just fine. When we put myTextStream.Writeline("Fred") the actual file gets two extra numbers written to it in addition to those representing F·r·e·d. These are called "Carriage Return" and "Line Feed". They get their names from type writers. They represent the action of returning the carriage (holding the paper) to the left and feeding through the paper by one line height. Nowadays these numbers just represent the end of a line and the beginning of the next.

When we use myTextStream.ReadLine() we get back the next line of text from a file. Just like a human reading a page of text, the TextStream Object remembers its 'place' in the text and so can give us the next line from that.

There are a few other things which we might want to do with a text file. These are not useful quite as often as working with one line at a time, but they come in handy every so often:

1) Read an entire file at once to a String.
2) Write an entire file at once from String.
3) Read an exact number of characters from a file.
4) Write a String to a file without adding on the carriage-return line-feed.

Whilst these are four different things we need only learn two new techniques to be able to do them! When we open a TextStream for reading we can call call myTextStream.Read(*count*) where *count* is a number. This is just like Readline in that the TextStream Object remembers its 'place' in the file and reads on from that place. However, rather then reading a whole line, it reads exactly *count* characters and returns these as a String. Another slight difference is that when it encounters carriage-return or line-feed, it will actually put these characters in the returned String. Also note that if it reaches the end of the file before it has read *count* characters it just gives us the characters it has read so far.

In Excel Macro Scripting I have yet to come across a need to read *count* characters from a file except in one common special case. Sometime we want to read an entire file into a String all in one go. Again, to make this super easy, we can just miss out *count* altogether. In the below script we open a TextStream reading from a file and call ReadAll() which is like using Read(*count*) where *count* is the number of characters left which can be read. This slurps the entire file's contents into a String for us (this is not an entire script, just a piece of one):

```
  Set myFSO=CreateObject("Scripting.FileSystemObject")
  Set myTS=myFSO.OpenTextFile _
    ( _
      "myFile.txt", _
      forReading, _
      FALSE _
    )
  ' Read the entire contents at once
  fileContents=myTS.ReadAll()
  myTS.Close
```

The exact same thing can be done with writing. By using the Method Write(*myString*) on a TextStream Object (which was opened to write or append) we simply write out the numbers for the characters in the String to the file. Write(*myString*) does not add carriage-return or new-line. If we call it many times we can add to the files one chunk at a time. If we open a TextStream for overwriting (not appending) and we call Write(*myString*) just the once before closing the TextStream, we have written the entire contents of the file from our String.

Here is a script which reads an entire text file in one go. It then replaces every instance of "/" with "//". Finally, it writes a new file with the updated contents. We will see in lesson 7 (the next) why we might want to do this (again, this is just a subsection of a script):

```
  Set myFSO=CreateObject("Scripting.FileSystemObject")
  Set myTS=myFSO.OpenTextFile _
    ( _
      "myFile.txt", _
      forReading, _
      FALSE _
    )
  ' Read the entire contents at once
  fileContents=myTS.ReadAll()
  myTS.Close

  ' Do the replace of / with //
  fileContents=Replace(fileContents,"/","//")

  ' Now write out our new file contents
  Set myTS=myFSO.OpenTextFile _
    ( _
      "myFile.txt", _
      forWriting, _
      TRUE _
    )
  ' Write the entire contents at once
  myTS.Write(fileContents)
  myTS.Close
```

Listing Files

I was considering not including this section. Then I suddenly remembered a very important sort of application for listing files. Let us imagine that we have written a script which takes a 'drag-and-drop' Workbook and merges the data with a second Workbook. This script then creates an output Workbook containing the merged data (see lesson 5). Whilst this is fine for one, two or even ten input Workbooks, how about if there were 100 or even 1000. The alternative to spending the balance of our natural life dragging and dropping is to get our script to find the input Workbook files its self.

The next script does exactly that. It is a bit 'dense' in that it does a lot with only a few lines of code. Rather than stuff it full of comments, I have discussed how it works after the script.

```
Option Explicit
Dim fso,myFolder,myFile,myShell,myList

Set myShell=CreateObject("WScript.Shell")
Set fso=CreateObject("Scripting.FileSystemObject")
Set myFolder =
fso.GetFolder(myShell.SpecialFolders.Item("Desktop"))
myList=""
For Each myFile In myFolder.Files
  With myFile
    If _
      LCase(Right(.name,4))=".xls" OR _
      LCase(Right(.name,5))=".xlsx"   _
    Then _
      myList=myList & .name & vbcrlf
    End IF
  End With
Next
WScript.Echo "Excel Files On The Desktop:" & vbcrlf &
myList
```

The first thing that the above script does is to use Special Folders to find the path to the Desktop. Next it calls the GetFolder(*path*) method of a FileSystemObject Object. This method returns a Folder Object. Because the path we pass to the GetFolder method is the path to the GetFolder method, we get a Folder Obejct representing the Desktop.

Folder Objects are used to allow us to do things with Folders. In this case, we want to find all the files in the Folder. The Folder Object has a Collection in it called 'Files' which contains a File Object representing each of the Files in the Folder. This being a Collection is very useful because we can use a For Each loop to work through each Item in the Collection.

Inside the For Each loop I have placed an If statement. This is what is called a 'conditional'. What this means is that If a condition is met

Then a piece of code is executed. To try and explain this further I can take the piece of script...

```
    With myFile
      If _
        LCase(Right(.name,4))=".xls" OR _
        LCase(Right(.name,5))=".xlsx"   _
      Then _
        myList=myList & .name & vbcrlf
      End If
    End With
```

..and try and describe it as in plain English:

> If the *name* of the Variable myFile ends with ".xls" or with ".xlsx" then add that name to the end of the Variable myList. Otherwise, do not do anything.

We have already see the VBScript built in function LCase. Another handy built in function is Right. This function returns just the Right hand characters from a String. Appendix D has a description of all these built in functions. VBCRLF is a built in 'Constant'. It contains a String consisting of carriage-return, line-feed. By adding this to the end of each File name as it is added to myList, it makes the names appear on new lines in the popup box created at the end of the script.

To process all the files of a particular type in a folder all we need to do is replace the code...

```
      If _
        LCase(Right(.name,4))=".xls" OR _
        LCase(Right(.name,5))=".xlsx"   _
      Then _
        myList=myList & .name & vbcrlf
      End IF
```

...with what ever code we require. For example, our Workbook merging code.

'Take Home' Ideas from Lesson 6

Files are just simple lists of numbers.

By assigning characters to the numbers we can store text in files. We call these "text files".

The TextStream Object does the conversion from text to numbers (and back) so it lets us read an write text files using Strings.

By adding more rules about the order of data in files, more complex information can be stored in them. We call these structure rules "format".

When writing files we can overwrite older versions of the file or append new stuff to then.

Most reading and writing is done one line at a time.

Being able to read and write files allows our scripts to work with file formats which Excel does not understand by its self.

Sometimes it is handy to read a whole file into a String or write a whole file out from a String.

We can list all the files in a folder so that we can then process them all, or a subset of them, automatically.

Lesson 7: Enhanced Data Processing

Subroutines And Functions

Before going further, let us understand one thing. Subroutines and Functions are here to make our life easier. They are not some abstract concept which computer scientists thrust upon us to prove a point; they are here only for our convenience.

The whole concept starts of from a simple idea: *It is best to do one thing one time; it is a waste of effort to do one thing many times.*

This applies double or triple to writing Scripting Macros. If we write the same piece of code twice we have doubled the input effort. Then, if we need to change that code, we have to remember all the places we put it and change each and then fix any typos we have made!

So, write one thing one time; it is much easier.

A Subroutine allows us to write a piece of script once but use it many times. For example, we might want to make a piece of script which turns the background of some Cells yellow. To do this as a Subroutine we can write:

```
Sub MakeYellow(myRange)
  With myRange.Interior
    .ColorIndex = 6
    .Pattern = 1
    .PatternColorIndex = -4105
  End With
End Sub
```

Functions are a lot like Subroutines except that they give a value. Here is a stupidly simple Function:

```
Function AddTwoNumbers(a,b)
  AddTwoNumbers=a+b
End Function
```

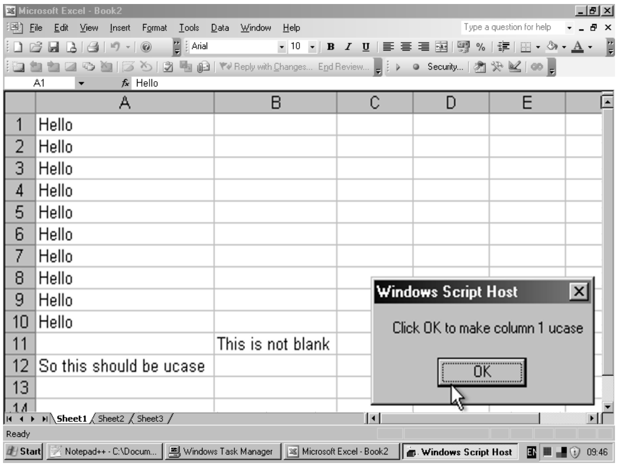
By making the name of the Function equal to the thing we want given back, we control what it gives back. The above Function could be used like this:

```
Option Explicit
WScript.Echo AddTwoNumbers(2,2)
WScript.Echo AddTwoNumbers(3,4)
Function AddTwoNumbers(a,b)
  AddTwoNumbers=a+b
End Function
```

This script will produce two pop-up boxes; the first will say “4” and the second “7”.

Here is a more useful Function. It looks down a Column and turns every Cell Value into upper case. It will stop when it finds an entire Row which is empty, returning the number of altered Rows:

```
 Option Explicit
 Dim myExcel, myWorkbook,mySheet,col,row

 Set myExcel=CreateObject("Excel.Application")
 Set myWorkbook=myExcel.Workbooks.Add()
 Set mySheet=myWorkbook.Sheets(1)
 myExcel.Visible=true

 ' Create some data to demonstrate the Function
 For row=1 to 10
   mySheet.Cells(row,1)="Hello"
 Next
 mySheet.Cells(11,2)="This is not blank"
 mySheet.Cells(12,1)="So this should be ucase"
 mySheet.Columns(1).Autofit
 mySheet.Columns(2).Autofit

 ' These three lines do all the work, they
 ' use pop-ups to tell us what is happening and
 ' call the Function
 WScript.Echo "Click OK to make column 1 ucase"
 row=UCaseCol(mySheet,1)
 WScript.Echo row & " Rows were converted"

 ' This is the actual Function its self.
 Function UCaseCol(theSheet,theCol)
   Dim uRow,uCol
   ' The max number of rows could be bigger
   ' in Excel 2007 but in all previous versions
   ' it is 65536, so we will use that
   For uRow=1 to 65536
     ' The max number of columns could bigger
     ' in Excel 2007 but in all previous versions
     ' it is 256, so we will use that
     For uCol=1 to 256
       If Not IsEmpty(theSheet.Cells(uRow,uCol).Value)
Then
         ' Exit For forces the loop to stop
         ' in this case, it will stop the
         ' "For uCol" loop
         Exit For
       End If
     Next
   ' If uCol=257 here then all Cells were empty
   If uCol=257 Then
    ' This Exit For will case the "For uRow"
    ' loop to stop
   Exit For
   End If
   With theSheet.Cells(uRow,theCol)
    .Value=UCase(.Value)
   End With
  Next

  ' At this point uRow is always the first blank row
  ' but we want to give back the number of
  ' changed rows
  UCaseCol=uRow-1

 End Function
```



	A	B	C	D	E
1	Hello				
2	Hello				
3	Hello				
4	Hello				
5	Hello				
6	Hello				
7	Hello				
8	Hello				
9	Hello				
10	Hello				
11		This is not blank			
12	So this should be ucase				
13					

Windows Script Host

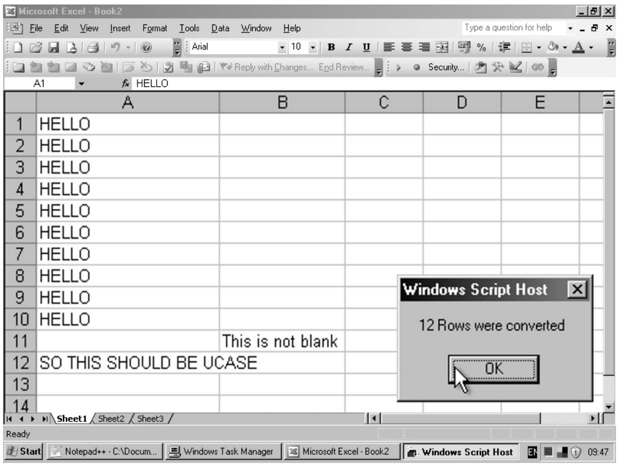
Click OK to make column 1 ucase

OK

Figure X: Here is what the "upper-case a Column" script produces at the point where the first pop-up appears.

It is important to remember that is we create a Function which gives back (sometimes called 'returns') an Object, we must use Set:

```
 ' A Function to create a new Excel Object and
 ' return a new Workbook from it
 Function MakeExcelWorkbook()
  Dim myExcel
  Set myExcel=CreateObject("Excel.Application")
  Set  MakeExcelWorkbook=myExcel.Workbooks.Add()
 End Function
```



A1 HELLO

	A	B	C	D	E
1	HELLO				
2	HELLO				
3	HELLO				
4	HELLO				
5	HELLO				
6	HELLO				
7	HELLO				
8	HELLO				
9	HELLO				
10	HELLO				
11		This is not blank			
12	SO THIS SHOULD BE UCASE				
13					
14					

Windows Script Host

12 Rows were converted

Figure X: Here is what the "upper-case a Column" script produces once it has finished processing the Sheet.

Death By Dates

I have been working in IT for what feels like a very long time (over 25 years – sob). Over that time, certain patterns have started to appear to me. One of these is that when a data set goes all weird, it is usually because of dates.

The reason is simple; dates have no consistently applied standards. We can illustrate this by counter example. Integer numbers do have a very consistent standard. Nearly all integer number in data are represented using Arabic numeral base ten notation:

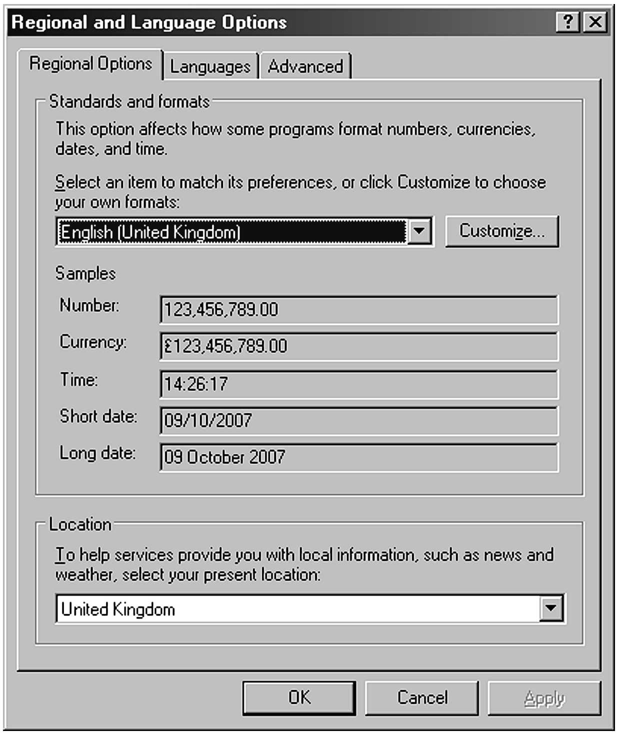
- `1`
- `11`
- `100`
- `-20`

By way of contrast, here are just a few of the ways one might see "the first of July two thousand and one" written:

- `1/7/01`
- `1/7/2001`
- `7/1/01`
- `1-7-01`
- `7/1/2001`
- `1-July-2001`

Here is a good question: “If 7/1/01 and 1/7/01 mean exactly the same thing, then how can we tell they represent the first of July or the seventh of January?”

The problem with Excel and dates usually turns up when importing data where the source of the dates or the Excel Spreadsheet are set to be 'outside' the USA. The snag is that even if we set the Windows local to be somewhere where dates are dd/mm/yyyy (day day/month month/year year year year as in 31/01/2002) Excel does not import such dates correctly. If you set the value of a Cell to 31/01/2001 it may well not recognise this as a date at all.



This option affects how some programs format numbers, currencies, dates, and time.

Select an item to match its preferences, or click Customize to choose your own formats:

English (United Kingdom)

Number:	123,456,789.00
Currency:	£123,456,789.00
Time:	14:26:17
Short date:	09/10/2007
Long date:	09 October 2007

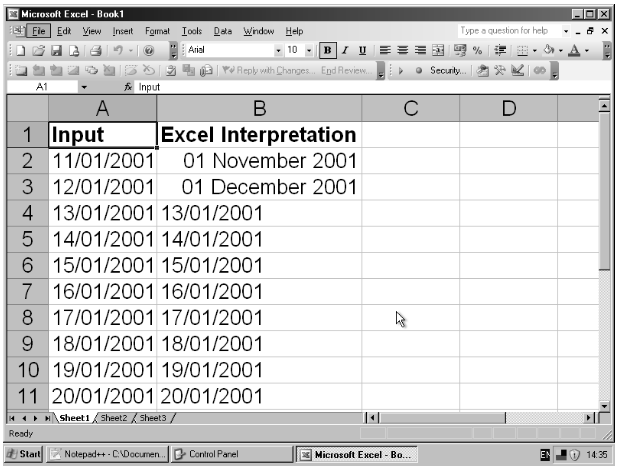
To help services provide you with local information, such as news and weather, select your present location:

United Kingdom

Figure 20: Here is the language and regional configuration of my test (virtual) computer whilst writing 'Baby Steps'.

To illustrate this issue, I have created a script which loads Excel with dates from the 10th to the 20th of January. It also uses a trick to force Excel not it interpret the values I am passing into it. This way I can get Excel to show what the 'raw' value I put in it and then also what it interprets it as. This trick will be used again later.

```
  Option Explicit
  Dim myExcel,myWorkbook,mySheet,row
  Set myExcel    = CreateObject("Excel.Application")
  Set myWorkbook = myExcel.Workbooks.Add()
  Set mySheet    = myWorkbook.Sheets(1)

  ' This loop creates strings in dd/mm/yyyy format
  ' puts them into Cells.  To force Excel into not
  ' interpreting the string in any way, I make it
  ' into a formula and use the Cell.Formula property
  ' rather than the Cell.Value property we normally
  ' use.
  For row=1 To 10
    ' This sets a formula like ="11/01/2001"
    mySheet.Cells(row+1,1).Formula = "=""" & (10+row) &
"/01/2001"""
    ' This will just put in the date string
    ' like 11/01/2001 and it also sets the Cell
    ' format to show dates in long format to help
    ' make it clear what Excel's interpretation is
    With mySheet.Cells(row+1,2)
      .Value = "" & (10+row) & "/01/2001"
      .NumberFormat = "[$-F800]dddd, mmmm dd, yyyy"
    End with
  Next
  ' This code puts in the column titles and makes
  ' them bold
  mySheet.Cells(1,1).Value="Input"
  mySheet.Cells(1,2).Value="Excel Interpretation"
  mySheet.Range("A1:B1").Font.Bold = True
  ' Now we can autofit the width of the columns
  ' and as a nice touch, zoom into the spreadsheet
  mySheet.Columns(1).Autofit
  mySheet.Columns(2).Autofit
  myExcel.ActiveWindow.Zoom = 200
  ' Finally, we had better make Excel visible!
  myExcel.Visible = TRUE
```



	A	B
1	Input	Excel Interpretation
2	11/01/2001	01 November 2001
3	12/01/2001	01 December 2001
4	13/01/2001	13/01/2001
5	14/01/2001	14/01/2001
6	15/01/2001	15/01/2001
7	16/01/2001	16/01/2001
8	17/01/2001	17/01/2001
9	18/01/2001	18/01/2001
10	19/01/2001	19/01/2001
11	20/01/2001	20/01/2001

Figure 21: Here we can see how Excel has made a complete mess of interpreting the values which our script as placed into it.

DateAdd And DateSerial – Fixing The Problem

First we can look at a VBScript built in function called 'DateAdd'. We can use this to bypass the faulty date interpretation in Excel.

```
DateAdd(Interval, Number, Date)
```

The DateAdd function adds a time interval to any date. There are three arguments. The Interval argument defines the the type of time interval you wish to add onto the date. You must place the setting inside double quotes. The possible settings are shown in the table below:

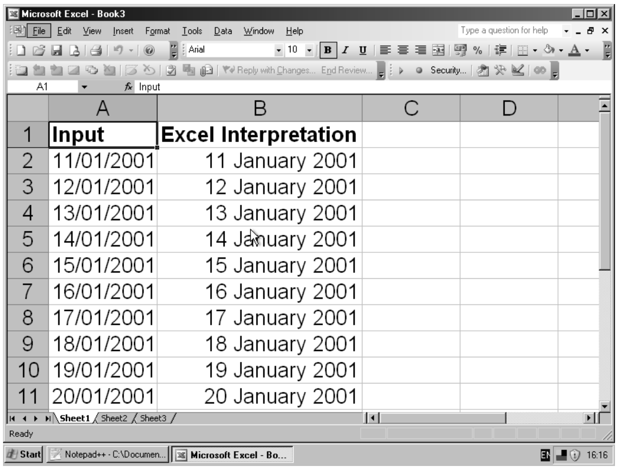
SETTING	DESCRIPTION
YYYY	Year
Q	Quarter
M	Month
Y	Day Of Year
D	Day
W	WeekDay
WW	Week Of Year
H	Hour
N	Minute
S	Second

The Number argument is a multiplier for the Interval argument (i.e., how many days, weeks, months, etc.). If this is a positive number, you will go forward in time. A negative number will go back in time.

The Date argument is an actual Date. This means it is not a String in date format like “01/02/2001”; it is a value of type Date. We can make Date values from strings using the VBScript built in function CDate.

```
  Option Explicit
  Dim
  myExcel,myWorkbook,mySheet,year,month,day,myDate,start
Date,row
  Set myExcel    = CreateObject("Excel.Application")
  Set myWorkbook = myExcel.Workbooks.Add()
  Set mySheet    = myWorkbook.Sheets(1)

  ' This date cannot be confused because the day
  ' and month are the  same, and it is a nice
  ' simple date.
  ' We will use it as the starting date from which to
  ' make our calculations
  startDate=CDate("01/01/2000")

  For row=1 to 10
    ' Here we set up the day month and year
    myMonth=1
    myYear=2001
    myDay=row+10

    ' Write the 'raw' date string into Excel
    mySheet.Cells(row+1,1).Formula = "=""" & myDay &
"/01/2001"""

    ' Compute a true Date value for the date
    ' We start of with the startDate
    myDate=startDate
    ' Then add in the appropreate number of years
    ' It does not matter if year < 2000, because this
    ' will still work OK
    myDate=DateAdd("YYYY",myYear-2000,myDate)
    ' Now we add in the appropreate number of months
    myDate=DateAdd("M",myMonth-1,myDate)
    ' And finally the appropreate number of days
    myDate=DateAdd("D",myDay-1,myDate)

    ' At this point myDate holds a Date type value
    ' that represents the date we are interested in.
    ' Excel does not need to interpret this value,
    ' because it is already a date, so Excel cannot
```

```
        ' make a mistake.
        With mySheet.Cells(row+1,2)
            .Value = myDate
            .NumberFormat = "[$-F800]dddd, mmmm dd, yyyy"
        End With
    Next

    ' This is just the same layout stuff as before
    mySheet.Cells(1,1).Value="Input"
    mySheet.Cells(1,2).Value="Excel Interpretation"
    mySheet.Range("A1:B1").Font.Bold = True
    mySheet.Columns(1).Autofit
    mySheet.Columns(2).Autofit
    myExcel.ActiveWindow.Zoom = 200
    myExcel.Visible = TRUE
```



	A	B	C	D
1	**Input**	**Excel Interpretation**		
2	11/01/2001	11 January 2001		
3	12/01/2001	12 January 2001		
4	13/01/2001	13 January 2001		
5	14/01/2001	14 January 2001		
6	15/01/2001	15 January 2001		
7	16/01/2001	16 January 2001		
8	17/01/2001	17 January 2001		
9	18/01/2001	18 January 2001		
10	19/01/2001	19 January 2001		
11	20/01/2001	20 January 2001		

Figure 22: Here we can see the new script correctly creates date values in the Excel Cells.

Another (possibly simpler yet less informative) way of achieving the same thing is to use the DateSerial built in function. This takes 3 arguments, the year, month and day. So the previous script piece:

```
        myDate=startDate
        myDate=DateAdd("YYYY",myYear-2000,myDate)
        myDate=DateAdd("M",myMonth-1,myDate)
        myDate=DateAdd("D",myDay-1,myDate)
```

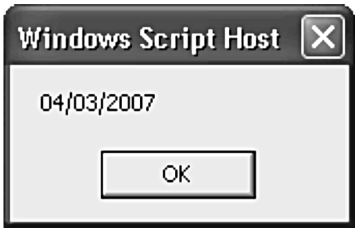
Becomes:

```
        myDate=DateSerial(year,month,day)
```

I considered only including this simpler version. However, the more complex DateAdd version made such a nice example of date arithmetic, I chose to include both approaches. Indeed, the the DateAdd method is illustrative of the way DateSerial actually works. For example, what does the following script produce?

```
Option Explicit
Dim myDate
myDate=DateSerial(2007,2,32)
WScript.Echo myDate
```

What it comes up with is a pop up box giving the fourth of March as a date.



```
04/03/2007
```

Figure 23: This is the popup showing that DateSerial actually works just like DateAdd. Here, DateSerial(2007,2,32) yields the fourth of March, i.e. two months and 32 days added to 2007.

The fourth of March being the Date you would get if you were to do this:

```
myDate="1/1/2007"
myDate=DateAdd("M",2-1,myDate)
myDate=DateAdd("D",32-1,myDate)
WScript.Echo myDate
```

Importing Data From CSV Files

We looked at file formats in lesson 6. The idea of a format being that the order of numbers in a file offers information as to the meaning of those numbers. If the file is a text file, then we can consider the order of numbers to represent an order of characters. There are some text file formats that are used a very great deal. One of the oldest yet still most commonly used ones is “Comma Separated Values” or CSV. Files which contain text in CSV format are called CSV files and they turn up everywhere.

We come across this standard format most when we are transfering data from one application to another. For example, if a customer relationship management system must create an output which we can read with Excel, CVS might be an excellent choice:

Customer	Call Length	Satisfaction
Fred F.	11	***
Barny R.	15	**
Scoobi D.	5	*****
Arther D.	25	*

→

```
Customer,Call Length,Satisfaction
Fred F.,11,***
Barny R.,15,**
Scoobi D.5,*****
Arther D.,25,*
```

Figure 24: This shows how a database view from a CRM (Customer Relationship Management) system can be converted to a CSV file format.

In the above example a view in a CRM system has been exported to a CSV file. Excel is able to read CSV files and convert the contents directly into a spreadsheet. There are even some cleaver standards to avoid things like commas in the data causing trouble. If there was a comma in the data, the field holding the comma is surrounded in quotation marks:

Customer	Call Length	Satisfaction
Fred, F.	11	***

→

```
Customer,Call Length,Satisfaction
"Fred, F.",11,***
```

Figure X: Here we can see how when a data field has a comma in it, the CSV format places that field in quotation marks to make it clear which commas are separating fields and which are contained inside fields.

Unfortunately, CSV tells us nothing about the nature of the data, it just has columns and rows. However, Excel tries to guess the nature of data from its format. As we have already seen, Excel often takes something like this “5/7/01” and treats it as a date. But it also gets all confused between the fifth of July and the seventh of May.

Because of these issues, it is sometimes necessary to 'help' Excel along a bit when reading CSV files. This can be done with a three step strategy:

1) Change the data in the CSV file so that Excel will treat everything as simple text.
2) Open Excel and tell it to load the altered CSV file.
3) Change the data in Excel to the format we want or insert formulae which output the format we want (or both).

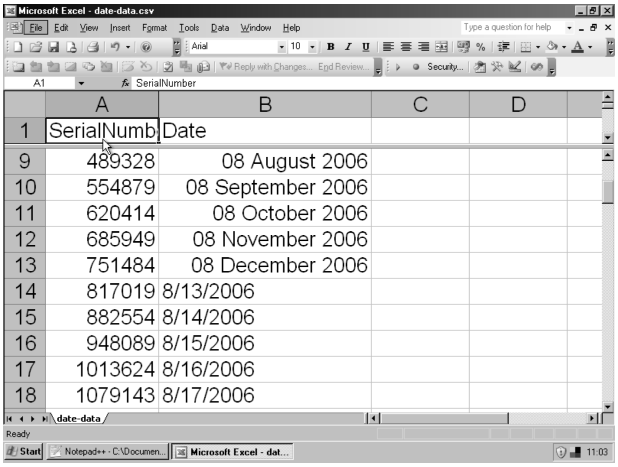
For step 1 we can use the file handling techniques we looked at the previous lesson. We open the original CSV file as a TextStream Object and open an empty output CSV file as another TextStream. We can the read the original in, alter the contents and write the new content out.

To illustrate this survival trick I will go through two examples; one is for working with dates and the for working with things that Excel might incorrectly think are numbers. For example, if we type “10e24” into an Excel Cell, Excel will convert this to 1.00E25; in other words, Excel has read the text and considered it to represent 1 x 10^{24}.

```
 ' Create a CSV file with a load of dates in it.
 ' This script is here
 ' only to produce the input file for the next
 ' script – it is not really part of the example
 Option Explicit
 Dim myFSO, myTS,row
 Const forWriting = 2
 Set myFSO=CreateObject("Scripting.FileSystemObject")
 Set myTS=myFSO.OpenTextFile("Date-
data.csv",forWriting,TRUE)
 myTS.WriteLine "SerialNumber,Date"
 For row=0 to 99
   ' This is a numerical trick to produce realistic
   ' serial numbers from the row Variable and dates
   ' in US format which a UK Excel setting will get
   ' wrong.
   myTS.WriteLine (NOT ((row + row * 65536) XOR
&H8888)) & ",8/" & (1+row-(31*Int(row/31))) & "/2006"
 Next
 myTS.Close
```

Here is what the contents of the created CSV file look like:

```
SerialNumber,Date
30583,8/1/2006
96118,8/2/2006
161653,8/3/2006
227188,8/4/2006
292723,8/5/2006
358258,8/6/2006
```


Microsoft Excel - date-data.csv
File Edit View Insert Format Tools Data Window Help
A1 SerialNumber
date-data
Ready
Start
Notepad++ - C:\Documen...
Microsoft Excel - dat...
11:03

	A	B	C	D
1	SerialNumb	Date		
9	489328	08 August 2006		
10	554879	08 September 2006		
11	620414	08 October 2006		
12	685949	08 November 2006		
13	751484	08 December 2006		
14	817019	8/13/2006		
15	882554	8/14/2006		
16	948089	8/15/2006		
17	1013624	8/16/2006		
18	1079143	8/17/2006		

Figure 26: This is how Excel interprets the CSV file which our script creates. Clearly, it has read the dates totally incorrectly.

```
' Change the CSV file innards to force Excel
' to treat them as plain text
Option Explicit
Dim
myExcel,myWorkbook,mySheet,row,myValue,myFSO,myTS,file
Contents,myShell,myDesktop,splitted
Const forReading=1
Const forWriting=2

Set myShell=CreateObject("WScript.Shell")
myDesktop=myShell.SpecialFolders("Desktop")

Set myFSO=CreateObject("Scripting.FileSystemObject")
Set myTS=myFSO.OpenTextFile(myDesktop & "/Date-
data.csv",forReading,FALSE)
fileContents=myTS.ReadAll()
myTS.Close
fileContents=Replace(fileContents,"/","//")
Set myTS=myFSO.OpenTextFile(myDesktop & "/Date-data-
fixed.csv",forWriting,TRUE)
myTS.Write(fileContents)
myTS.Close
```

The above script work by replacing every occurrence of / with //. This means that a date String “1/5/01” will become “1//5//05”. Here is a piece of the resulting file:

```
SerialNumber,Date
30583,8//1//2006
96118,8//2//2006
```

```
    161653,8//3//2006
    227188,8//4//2006
    292723,8//5//2006
    358258,8//6//2006
```

Whilst this has stopped Excel incorrectly interpreting dates, it has not performed step 3 in our strategy. To do that we need to go through all the fields that should be dates in Excel and convert them into Excel correct dates. Not only that, we also need to correct all text fields to undo any changes the previous step caused. For example, "the choice was green/blue" will have been changed to "the choice was green//blue". These fields need changing back. The following script performs this action on our example file. It is actually a continutation of the previous script. It makes sense to combine the file correction and loading into one Scripting Macro:

```
  ' Read the newly updated CSV file into Excel and
  ' then recreate the dates in a way Excel cannot
  ' get confused (see previous lesson)
  Set myExcel=CreateObject("Excel.Application")
  myExcel.Visible=TRUE
  ' Load the csv usign the Open Method of the
  ' Workbooks Collection
  Set myWorkbook=myExcel.Workbooks.Open(myDesktop &
"/Date-data-fixed.csv")
  Set mySheet=myWorkbook.Sheets(1)
  ' This loop goes down the Rows from the first data
  ' until the first Row in which the "serial number"
  ' is blank
  For row=2 to 65536
    If IsEmpty(mySheet.Cells(row,1).Value) Then Exit For
    ' Here we use the ever handy Split built in Function
    ' to split the date String in Excel into
    ' three bits (0, 1 and 2)
    myValue=mySheet.Cells(row,2).Value
    splitted=Split(myValue,"//")
    ' DateSerial can now be used to create a real Date
    ' value to put back into Excel
    myValue=DateSerial _
      ( _
        splitted(2), _
        splitted(0), _
        splitted(1) _
      )
    mySheet.Cells(row,2).Value=myValue
    ' Finally, we correct the other Cell values
    myValue=mySheet.Cells(row,1).Value
    myValue=Replace(myValue,"//","/")
    mySheet.Cells(row,1).Value=myValue
  Next
```

Here is the same set of scripts for correcting the problems caused by scientific notation numbers:

```
  ' The script creates a csv file where the serial
  ' numbers are interpreted by Excel as scientific
  ' numbers
  Option Explicit
  Dim myFSO, myTS,row
  Const forWriting = 2
  Set myFSO=CreateObject("Scripting.FileSystemObject")
  Set myTS=myFSO.OpenTextFile("Serial-
data.csv",forWriting,TRUE)
  myTS.WriteLine "SerialNumber,Action"
  For row=1 to 100
    ' Here we use the same numerical trick to produce
    ' realistic looking serial numbers but with an 'e'
    ' in them so Excel misinterprets them as scientific
    ' notation
  myTS.WriteLine(NOT ((row + row * 65536) XOR &H8888)) &
"e" & row & ",Incomming"
  Next
  myTS.Close
```

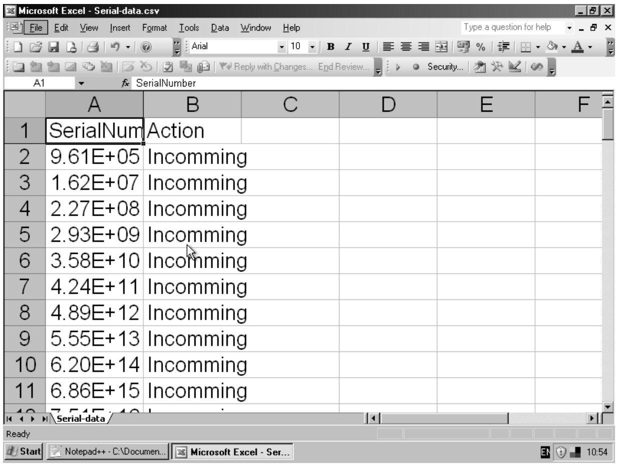
Here is what the contents of the created CSV file look like:

```
SerialNumber,Action
96118e1,Incomming
161653e2,Incomming
227188e3,Incomming
292723e4,Incomming
358258e5,Incomming
423793e6,Incomming
489328e7,Incomming
```

Here is the script to correct this behaviour:

```
  ' Change the CSV file innards to force Excel
  ' to treat them as plain text
  Option Explicit
  Dim
  myExcel,myWorkbook,mySheet,row,myValue,myFSO,myTS,file
  Contents,myShell,myDesktop,fixed
  Const forReading=1
  Const forWriting=2

  Set myShell=CreateObject("WScript.Shell")
  myDesktop=myShell.SpecialFolders("Desktop")
  Set myFSO=CreateObject("Scripting.FileSystemObject")
  Set myTS=myFSO.OpenTextFile(myDesktop & "/Date-
data.csv",forReading,FALSE)
  fileContents=myTS.ReadAll()
  myTS.Close
  fileContents=Replace(fileContents,"e","ee")
  Set myTS=myFSO.OpenTextFile(myDesktop & "/Date-data-
fixed.csv",forWriting,TRUE)
  myTS.Write(fileContents)
  myTS.Close

  ' Read the newly updated CSV file into Excel and
  ' then recreate the dates in a way Excel cannot
  ' get confused (see previous lesson)
  Set myExcel=CreateObject("Excel.Application")
  myExcel.Visible=TRUE
  ' Load the csv usign the Open Method of the
  ' Workbooks Collection
  Set myWorkbook=myExcel.Workbooks.Open(myDesktop &
"/Date-data-fixed.csv")
  Set mySheet=myWorkbook.Sheets(1)
  ' This loop goes down the Rows from the first data
  ' until the first Row in which the “serial number”
  ' is blank
  For row=2 to 65536
    If IsEmpty(mySheet.Cells(row,1).Value) Then Exit For
    ' This time we convert ee into e and the use the
    ' "Formula" techniques to force Excel not to
    ' interpret the serial number
    myValue=mySheet.Cells(row,1).Value
    fixed=Replace(myValue,"ee","e")
    mySheet.Cells(row,1).Formula = "=""" & fixed & """"
    ' Finally, we correct the other Cell values
    myValue=mySheet.Cells(row,2).Value
    myValue=Replace(myValue,"ee","e")
    mySheet.Cells(row,2).Value=myValue
  Next
```

Please note that in VBScript """" actually means a String containing just a quatation mark. So """Hello""" is a String "Hello". This is what the "=""" & Fixed & """" is doing; it is putting the String ="*nnnn*" into the Cell's formula where *nnnn* is the serial number.


Microsoft Excel - Serial-data.csv
File Edit View Insert Format Tools Data Window Help
Type a question for help
Arial
10
Reply with Changes... End Review...
Security...
A1
SerialNumber
Serial-data
Ready
Start
Notepad++ - C:\Documen...
Microsoft Excel - Ser...
10:54

	A	B	C	D	E	F
1	SerialNum	Action				
2	9.61E+05	Incomming				
3	1.62E+07	Incomming				
4	2.27E+08	Incomming				
5	2.93E+09	Incomming				
6	3.58E+10	Incomming				
7	4.24E+11	Incomming				
8	4.89E+12	Incomming				
9	5.55E+13	Incomming				
10	6.20E+14	Incomming				
11	6.86E+15	Incomming				

Figure 27: Again, Excel has incorrectly interpreted the contents of our CSV file. It has read the serial numbers, and because they have "e" in them, it has converted them into standard notation scientific numbers.

Creating Reports

Quite often Excel spreadsheets themselves make very good reports. However, sometime we want to write out reports in MS Word instead. There are a myriad of disturbingly complex ways of doing this. I, for one, like to avoid complex ways of doing anything so here is a nice simple way of creating reports in MS Word from Excel Macro Scripting.

The key to this method is using HTML. HTML being the way that web pages are created. Using HTML we can lay out text without using any wizardry. Really, it is very easy indeed to make tables and lists etc. For example:

```
<table border=1>
<tr>
<td>Hello</td><td>World</td>
</tr>
</table>
```

Makes something like this:

Hello	World

To make a list we can put:

```
<ul>
<li>This is the first list element.</li>
<li>This is the second list element.</li>
<li>This is the third list element.</li>
</ul>
```

This 'unordered list'will come out something like this:

- This is the first list element.
- This is the second list element.
- This is the third list element.

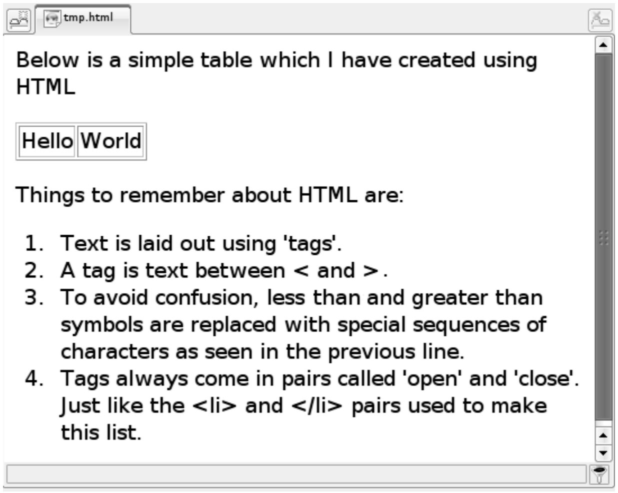
Text this is just normal paragraph writing can be laid out like this:

```
<p>This is paragraph text.  We do not want any special
layout for this.</p>
```

We can put all this together to make something like this:

```
<p>Below is a simple table which I have created using HTML</p>
<table border=1>
<tr>
<td>Hello</td><td>World</td>
</tr>
</table>
<p>Things to remember about HTML are:</p>
<ol>
<li>Text is laid out using 'tags'.</li>
<li>A tag is text between &lt; and &gt;.</li>
<li>To avoid confusion, less than and greater than symbols are
replaced with special sequences of characters as seen in the
previous line.</li>
<li>Tags always come in pairs called 'open' and 'close'. Just
like the &lt;li&gt; and &lt;/li&gt; pairs used to make this
list.</li>
</ol>
```

In this case I have created an “ordered list” by using “ol” rather tan the unordered list “ul”. If we save the above text in a file with called “text.html” we can read it with a web browser.



Below is a simple table which I have created using HTML

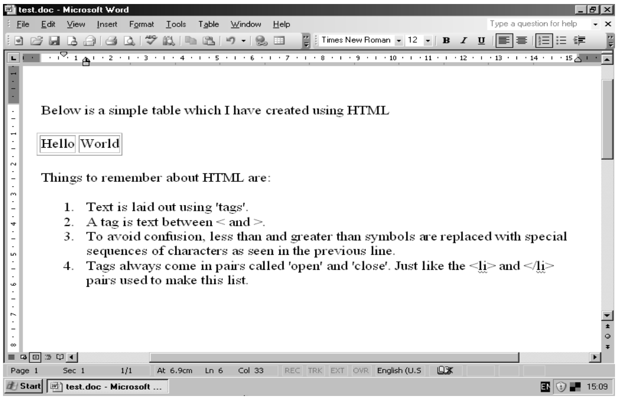
Hello	World

Things to remember about HTML are:

1. Text is laid out using 'tags'.
2. A tag is text between < and >.
3. To avoid confusion, less than and greater than symbols are replaced with special sequences of characters as seen in the previous line.
4. Tags always come in pairs called 'open' and 'close'. Just like the <li> and </li> pairs used to make this list.

Figure 28: Here is what a web browser displays when asked to open our simple "test.html" file. We can see that the special character sequence &lt; has been converted to < and &gt; has been converted to >. The tags inside the text like <li></li> etc. have caused the text to be laid out in the way we want.

This is all fine and wonderful, but we are supposed to be creating a MS Word report, not an HTML page. Well this is where a little known feature of MS Word comes into play. By simply saving the file as "test.doc" instead of "test.html" MS Word will magically reads it as a normal document. However, MS Word will still apply the HTML style layout rules.


test.doc - Microsoft Word
File Edit View Insert Format Tools Table Window Help
Type a question for help
Times New Roman 12
Page 1 Sec 1 1/1 At 6.9cm Ln 6 Col 33 REC TRK EXT OVR English (U.S
Start
test.doc - Microsoft ...
15:09

Below is a simple table which I have created using HTML

Hello	World

Things to remember about HTML are:

1. Text is laid out using 'tags'.
2. A tag is text between < and >.
3. To avoid confusion, less than and greater than symbols are replaced with special sequences of characters as seen in the previous line.
4. Tags always come in pairs called 'open' and 'close'. Just like the <li> and </li> pairs used to make this list.

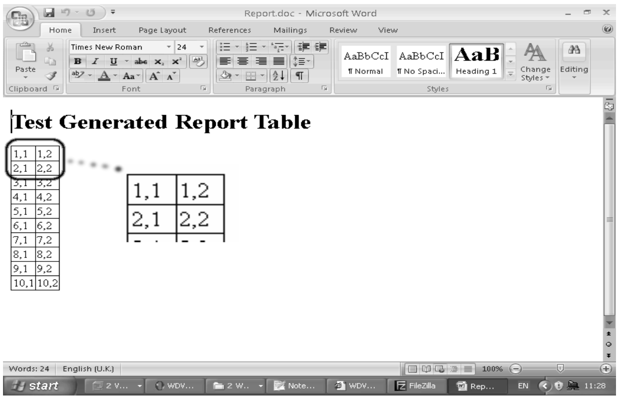
Figure 29: Here we can see the results of opening our simple HTML text as a MS Word document. I saved the HTML in a file called "test.doc" and then opened the file in MS Word.

So far we have looked at what happens with very simple HTML indeed. Nowadays HTML has grown to become a rich system for laying out text and images. MS Word does not really support all this complexity. However, we can enhance things a little. Also, it is probably about time we looked at a script creating the report!

The next script writes out a table in HTML into a '.doc' file. It uses a slightly more sophisticated way of defining the table border.

```
  myReportFile.WriteLine("<TABLE STYLE='border: 0px;
border-top: 1px solid black; border-left: 1px solid
black;' CELLSPACING=0>")
  myReportFile.WriteLine("<TBODY>")

  for row = 1 to 10
    myReportFile.WriteLine("<TR>")
    for column = 1 to 2
      myReportFile.WriteLine("<TD STYLE='border-bottom:
1px solid black; border-right: 1px solid black;'>" & row
& "," & column & "</TD>")
    next
    myReportFile.WriteLine("</TR>")
  next
  myReportFile.WriteLine("</TBODY></TABLE>")
  myReportFile.Close
```

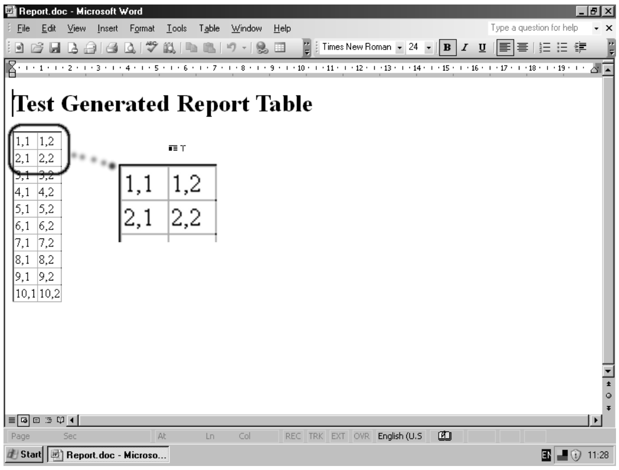
Whilst this script creates the data which it puts in the MS Word document, it could have equally well retrieved the data from an Excel Sheet.



Test Generated Report Table

1,1	1,2
2,1	2,2
3,1	3,2
4,1	4,2
5,1	5,2
6,1	6,2
7,1	7,2
8,1	8,2
9,1	9,2
10,1	10,2

Figure 30: This screen shot is of MS Word 2007 displaying our test report. The slightly more sophisticated description of the borders causes the nice single border rather than the default double border which we have seen in the earlier examples.


Report.doc - Microsoft Word
File Edit View Insert Format Tools Table Window Help
Type a question for help
Times New Roman 24
Start
Report.doc - Microso...
11:28

Test Generated Report Table

1,1	1,2
2,1	2,2
3,1	3,2
4,1	4,2
5,1	5,2
6,1	6,2
7,1	7,2
8,1	8,2
9,1	9,2
10,1	10,2

Figure 31: This screen shot is of MS Word 2003 displaying our test report. This older version of MS Word does a slightly less accurate job of interpreting the HTML; the border on the left and top of the table is thicker than it should be. I have included this image to illustrate that this technique is useful and easy but not 100% consistent across different versions of MS Office.

'Take Home' Ideas from Lesson 7

Re-writing the same piece of script over and over is a big waste of time.

Subroutines and Functions let us write a piece of script once but use it many times.

Functions are just like Subroutines except they give something back (they have a return value).

Data containing dates is often a cause of errors.

These errors are because date standards are applied inconsistently.

Reading a file, changing it to stop Excel making errors and writing it back out again can prevent Excel misinterpreting data like dates or serial numbers.

Replace is a VBScript built in function which replaces all the instances of one sub-string with another.

Lesson 8: Working With Pivot Tables

Why Pivot Tables Are So Very Important

Putting aside people skills, Pivot Tables are management embodied in a piece of software. To manage the supply of pumpkins one needs to know the trends and patterns in pumpkin deliveries; one does not need to know above every single pumpkin. Similarly, to manage hardware repair centres one needs to see what are the most common failures and how fast they tend to be fixed. Knowing that Jo Blogs has a modem that took 10 hours to fix might well be too much detail; knowing that, on average, 100 modems need fixing each day is critical management data.

A Pivot Table takes the 'too much detail' information and summarises it into something intelligible. For pumpkins, it could show the number of pumpkins arriving each day of the week. If it did this, it would be summarising 'count of pumpkins' by 'day of week'. Here we can start to see the core concept behind Pivot Tables. They summarise one or more data fields by some grouping field. This summary is performed using one of a number of available summary functions. Another example might be 'average % of pumpkins rotten on arrival' summarised by 'month of year'. There the data being summarised is 'rotten pumpkins on arrival'. The summary function is 'average' and the grouping field is 'month of the year'.

In short, Pivot Tables let one 'see' into a mass of detail and pick out critical management information. What is more, they then let us 'drill down' to see the underlying detail when we need it. We will see more of this super powerful drill-down ability later.

Understanding Pivot Tables

To help explain all the concepts behind Pivot Tables we are going to look at a fictional example of computer hardware repair times. In this example, there is a multi-national hardware company. The time taken to repair various broken components is measured in hours for the UK, US and China (Ch) offices. The management challenge here is that only raw data is available. What we start out with is just a Row in a spreadsheet for each repair. If we intend, for example, to improve the turn around for the slowest office, we will need to know which is the slowest office.

To throw extra fuel on this fire, we can pretend that there is a new company wide directive that repairs should be done in fourteen hours or less. How do we track the different offices' level of compliance and work out what is going wrong where compliance is not met? Well, compliance levels can be worked out directly using a Pivot Table. Then we can use that table to drill-down to the exact failing jobs numbers to then follow them up with a 'phone call to elucidate what is happening.

In short, we will use a Pivot Table to 'see' what is happening in the business at both an overview and detail view.

Creating An Example Pivot Table

Below is a script which 'manufactures' data. I have have done this so that there is no need to have a set of downloadable Workbooks to go with Baby Steps. Using a script like this is also a great way to develop any Excel Scripting Macro project. We can manufacture some nice clean data and work with that to create our script. Once our script works with clean data, we can try it with real world data. As we have seen earlier, real world data is 'dirty'; it has human errors in it. With this technique, we can add the ability to cope with dirty data into our Excel Scripting Macros in the sure knowledge that they work just fine with clean data.

```
Option Explicit
' Script to make some data for Pivot Table work
Dim myExcel,myWorkbook,mySheet
Dim row,rTime,wOffice,wEquipment, office, equipment
Dim r1,r2

Set myExcel=CreateObject("Excel.Application")
Set myWorkbook=myExcel.WorkBooks.Add()
Set mySheet=myWorkbook.Sheets(1)
myExcel.Visible=TRUE

mySheet.Cells(1,1).Value="Office"
mySheet.Cells(1,2).Value="Equipment"
mySheet.Cells(1,3).Value="Repair Time"
mySheet.Cells(1,4).Value="Job Number"

For row=2 To 301
  wOffice=Fix(rnd()*4+1)
  if wOffice = 1 then
    office="UK"
  elseif wOffice < 4 then
    office="US"
  else
    office="Ch"
  end if
  mySheet.Cells(row,1).Value=office
```

```
    wEquipment=Fix(rnd()*6+1)
    if wEquipment=1 then
      equipment="Modem"
    elseif wEquipment=2 then
      equipment="Hard Drive"
    elseif wEquipment=3 then
      equipment="Monitor"
    elseif wEquipment=4 then
      equipment="Keyboard"
    else
      equipment="Power Supply"
    end if
    mySheet.Cells(row,2).Value=equipment

    rTime=Fix(rnd()*10+1)+Fix(Rnd()*(10+wOffice)+1)
    mySheet.Cells(row,3).Value=rTime
    mySheet.Cells(row,4).Value=Fix(Rnd()*999+1000) & "-"
& Fix(Rnd()*999+1000)

  Next

  mySheet.Columns(1).Autofit
  mySheet.Columns(2).Autofit
  mySheet.Columns(3).Autofit
  mySheet.Columns(4).Autofit
```

How this script works:

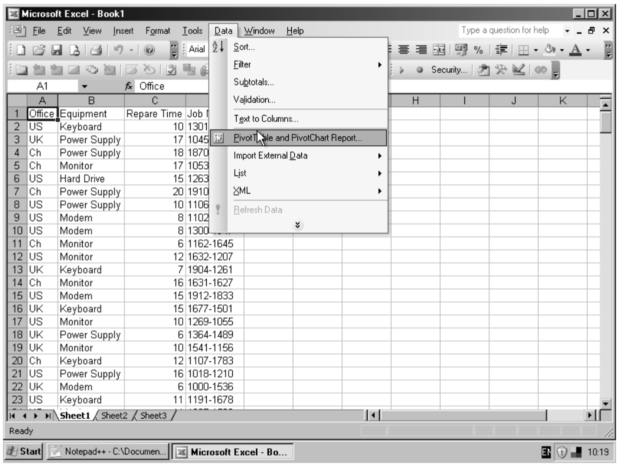
Fix(Rnd()*n+1) creates an integer number between 1 and n inclusive. Rnd() makes a random number between 0 and 1. Well, actually it is not truly random, it follows a mathematical pattern that appears to be random; this is called pseudo-random. Fix() takes any decimal and returns only the integer part. For example, if Rnd() gave 0.52 then Rnd()*10 would be 5.1 and Fix(Rnd()*10+1) would be 6.

We can use this formula to make a mix of at which office repairs are done, what is wrong and the job number. Also, if one takes two random numbers and adds them together, just like adding the number from throwing a pair of dice, the result will follow the classic Gaussian curve. That means the distribution of results is a bell shape with the bulk of results in the middle. I have used this technique to create a realistic distribution of repair times from the script.

Finally, to give us some variation between offices and between components which need repair, I have mixed in the numbers which choose office and device into the calculation of repair time. This means we get the type of variation that would be being looked for in a real situation.

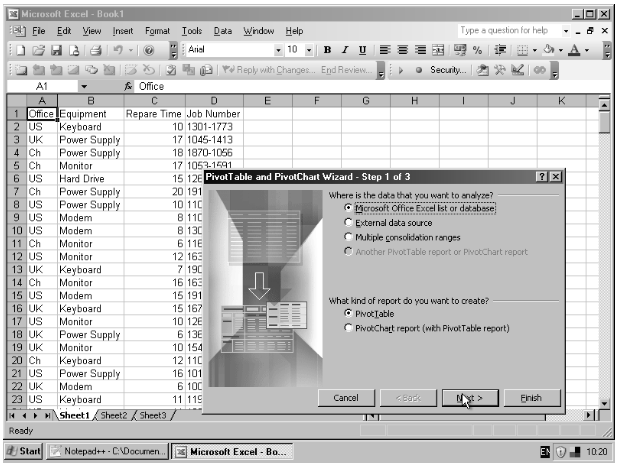
Creating A Pivot Table And Scripting Macro

Below are a set of screen shots which walk us through the process of creating a Pivot Table. The trick is to do this with the Macro Recorder running. Doing so will ensure Excel does all the hard work of creating a script for us; we need only translate and generalise the Macro it writes for us. I always start a new Pivot Table based project by Macro Recording myself creating the table using Excel's Pivot Table wizard; it is just so easy to use. In Appendix A, I have shown the same process but using Excel 2007. There are a few differences in the user interface but all the concepts are the same.



	A	B	C	
1	Office	Equipment	Repare Time	Job
2	US	Keyboard	10	1301
3	UK	Power Supply	17	1045
4	Ch	Power Supply	18	1870
5	Ch	Monitor	17	1053
6	US	Hard Drive	15	1263
7	Ch	Power Supply	20	1910
8	US	Power Supply	10	1106
9	US	Modem	8	1102
10	US	Modem	8	1300
11	Ch	Monitor	6	1162-1645
12	US	Monitor	12	1632-1207
13	UK	Keyboard	7	1904-1261
14	Ch	Monitor	16	1631-1627
15	US	Modem	15	1912-1833
16	UK	Keyboard	15	1677-1501
17	US	Monitor	10	1269-1055
18	UK	Power Supply	6	1364-1489
19	UK	Monitor	10	1541-1156
20	Ch	Keyboard	12	1107-1783
21	US	Power Supply	16	1018-1210
22	UK	Modem	6	1000-1536
23	US	Keyboard	11	1191-1678

Figure 32: To kick off the process of creating a Pivot Table, we must ensure that the Sheet with the data in which we are interested is showing. Then, selecting "Data" then "Pivot Table and Pivot Chart Report" will start the wizard we need to create out Pivot Table.



	A	B	C	D
1	Office	Equipment	Repare Time	Job Number
2	US	Keyboard	10	1301-1773
3	UK	Power Supply	17	1045-1413
4	Ch	Power Supply	18	1870-1056
5	Ch	Monitor	17	1053-1591
6	US	Hard Drive	15	126
7	Ch	Power Supply	20	191
8	US	Power Supply	10	110
9	US	Modem	8	110
10	US	Modem	8	130
11	Ch	Monitor	6	116
12	US	Monitor	12	163
13	UK	Keyboard	7	190
14	Ch	Monitor	16	163
15	US	Modem	15	191
16	UK	Keyboard	15	167
17	US	Monitor	10	126
18	UK	Power Supply	6	136
19	UK	Monitor	10	154
20	Ch	Keyboard	12	110
21	US	Power Supply	16	101
22	UK	Modem	6	100
23	US	Keyboard	11	119

PivotTable and PivotChart Wizard - Step 1 of 3

Where is the data that you want to analyze?
[✓ Microsoft Office Excel list or database]
External data source
Multiple consolidation ranges
Another PivotTable report or PivotChart report

What kind of report do you want to create?
[✓ PivotTable]
PivotChart report (with PivotTable report)

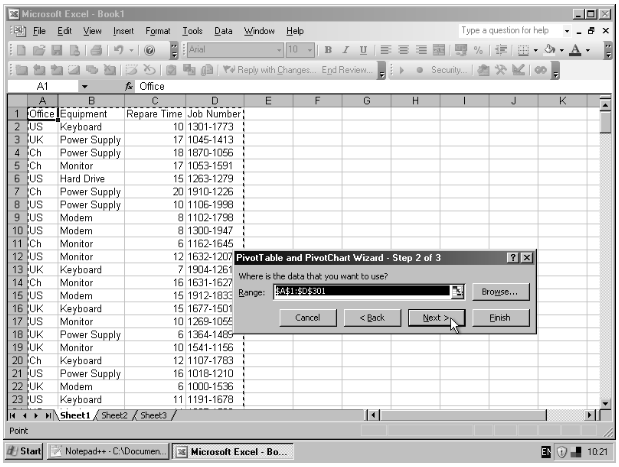
Figure 33: Next we will see the Pivot Table wizard start up. The important options on the first page of the wizard are where to get the data from and if the wizard should create a table or a chart.

Excel can create a Pivot Table or a Pivot Chart from external data. An example of this would be to create it from an SQL database. Whilst this is fabulously powerful, it is very rarely useful. For now we will choose to take the data from a Sheet. We can also ignore the option of using multiple consolidation ranges because, to be honest, it does not work very well.

I highly recommend always creating a Pivot Table first because we can always make a chart from it later. So the choices should be:

1) Microsoft Office Excel list or database.

2) Pivot Table.



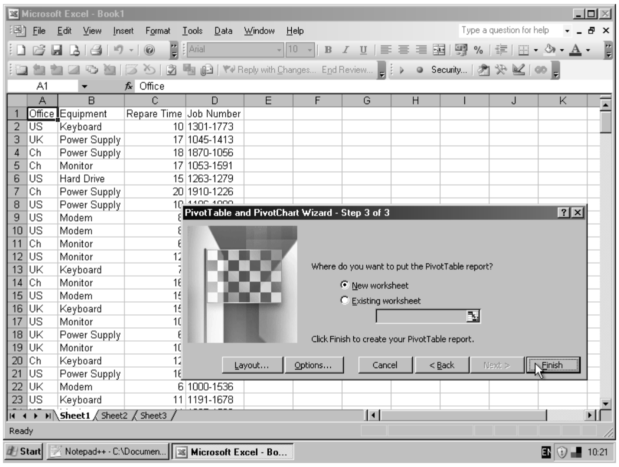
	A	B	C	D
1	Office	Equipment	Repare Time	Job Number
2	US	Keyboard	10	1301-1773
3	UK	Power Supply	17	1045-1413
4	Ch	Power Supply	18	1870-1056
5	Ch	Monitor	17	1053-1591
6	US	Hard Drive	15	1263-1279
7	Ch	Power Supply	20	1910-1226
8	US	Power Supply	10	1106-1998
9	US	Modem	8	1102-1798
10	US	Modem	8	1300-1947
11	Ch	Monitor	6	1162-1645
12	US	Monitor	12	1632-1207
13	UK	Keyboard	7	1904-1261
14	Ch	Monitor	16	1631-1627
15	US	Modem	15	1912-1833
16	UK	Keyboard	15	1677-1501
17	US	Monitor	10	1269-1055
18	UK	Power Supply	6	1364-1489
19	UK	Monitor	10	1541-1156
20	Ch	Keyboard	12	1107-1783
21	US	Power Supply	16	1018-1210
22	UK	Modem	6	1000-1536
23	US	Keyboard	11	1191-1678

PivotTable and PivotChart Wizard - Step 2 of 3

Where is the data that you want to use?

Range: $A$1:$D$301

Figure 34: Now that the wizard 'knows' that we want a Pivot Table we have to tell it from where to take the Sheet data. This is done using a standard Cell reference as show above. Excel usually makes the correct guess as to the Cells to use. If it gets it wrong, we can correct it by selecting the Cells with the mouse or typing the Cell reference into the wizard directly.



	A	B	C	D
1	Office	Equipment	Repare Time	Job Number
2	US	Keyboard	10	1301-1773
3	UK	Power Supply	17	1045-1413
4	Ch	Power Supply	18	1870-1056
5	Ch	Monitor	17	1053-1591
6	US	Hard Drive	15	1263-1279
7	Ch	Power Supply	20	1910-1226
8	US	Power Supply	10	
9	US	Modem	8	
10	US	Modem	8	
11	Ch	Monitor	6	
12	US	Monitor	12	
13	UK	Keyboard	7	
14	Ch	Monitor	16	
15	US	Modem	15	
16	UK	Keyboard	15	
17	US	Monitor	10	
18	UK	Power Supply	6	
19	UK	Monitor	10	
20	Ch	Keyboard	12	
21	US	Power Supply	16	
22	UK	Modem	6	1000-1536
23	US	Keyboard	11	1191-1678

PivotTable and PivotChart Wizard - Step 3 of 3

Where do you want to put the PivotTable report?

(● New worksheet) ○ Existing worksheet

Click Finish to create your PivotTable report.

Figure 35: Excel knows the basics of what we want at this stage. It knows we want a Pivot Table and what data will be used to create it. Next we must tell Excel where to put the new table. By far the tidiest solution is to always put a new Pivot Table in a new Sheet using the "New Worksheet" option.

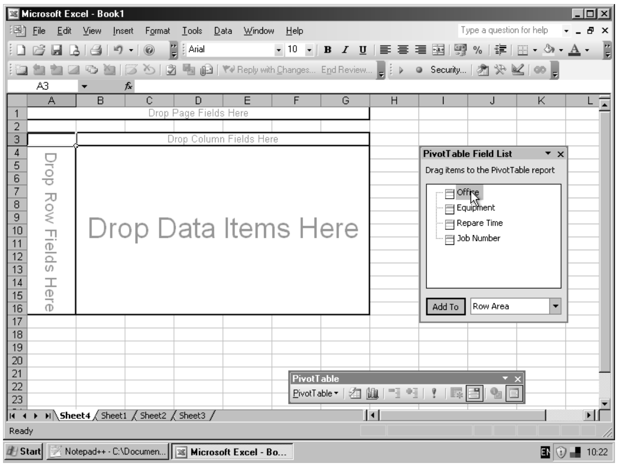


Figure 36: This screen shot shows the newly created but still blank Pivot Table and the Pivot Table Field List.

Now the fun starts! We can start laying out our Pivot Table. To facilitate this, Excel has drawn for use an empty Pivot Table. The table has 4 areas. This areas are 'drop targets'; we can drag field names from the Pivot Table Field List box into these areas of the Pivot Table to control the way the table works.

At the top of the blank table is the Page area; we will look at this later. Of more immediate interest is the Row area to the left of the table. This says “Drop Row Fields Here”. This area is for containing fields against which data will be summarised in Rows going across the table. Above and to the right of the Row area is the Column area saying “Drop Column Fields Here”. This area is for fields against which data will be summarised in Columns going down the page. In the middle is the Data area. This is where the data which is to be summarised is placed.

In our example we have repair times for different component failures at different offices. What we want to know is information on the repair time for a particular component at a particular office. We also want to be able to compare the over all performance of offices and the over all times for different components.

We can do all this by making the Pivot Table summarise by office in Rows and components in Columns. This way, to find the data for modems in the US office, we can simply look down the Rows until we find the US Row and then across the Columns until we find the modem Column. This will find us the Cell which will hold the number we are interested in. The Pivot Table will also give us a summary for each Row overall (for offices) and Column overall (for components). The next few screen shots show how we 'tell' the Pivot Table to do all these things.

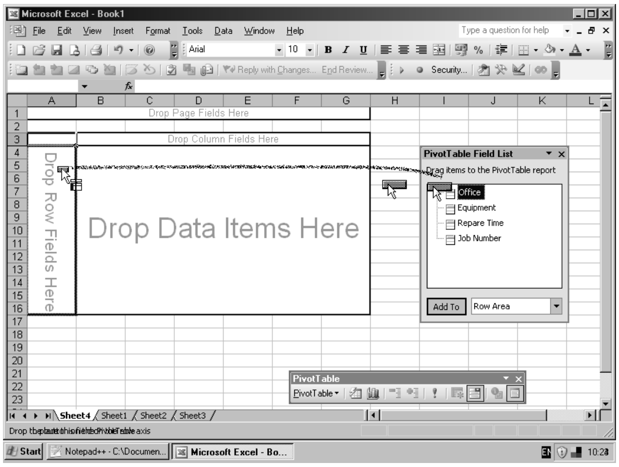
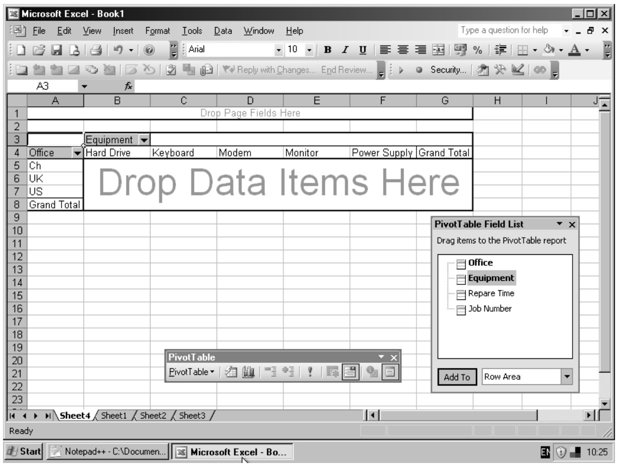


Figure 37: All the initial layout is done with drag-and-drop. The available fields are shown in the Pivot Table Field List dialogue box. To make "Office" a Row field we click on is name in the dialogue box and drag it over to the Row area where we drop it (as shown in the picture).

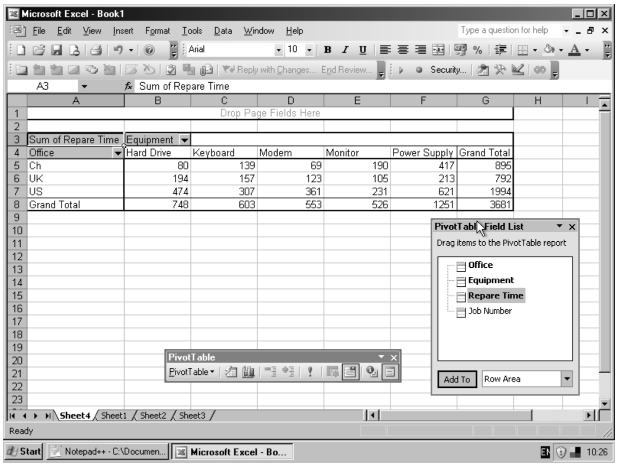
Dropping the "Office" field name onto the Row area immediately causes Excel to enumerate the unique values of this field as Row headers. It is also worth noting that should we change our minds, we can drag the grey "Office" button Excel adds at the top of the Row area. If we drag it back to the dialogue box from which it came, it is the "Office" field is removed from the Pivot Table. If we drag it to a different area (e.g. the Column area) Excel places it there instead.

Just like with the office field we can now drag-and-drop the "Equipment" field. This field represents the component that is being repaired. We drag it to the Column area and drop it there.



	Equipment					
Office	Hard Drive	Keyboard	Modem	Monitor	Power Supply	Grand Total
Ch						
UK						
US						
Grand Total						

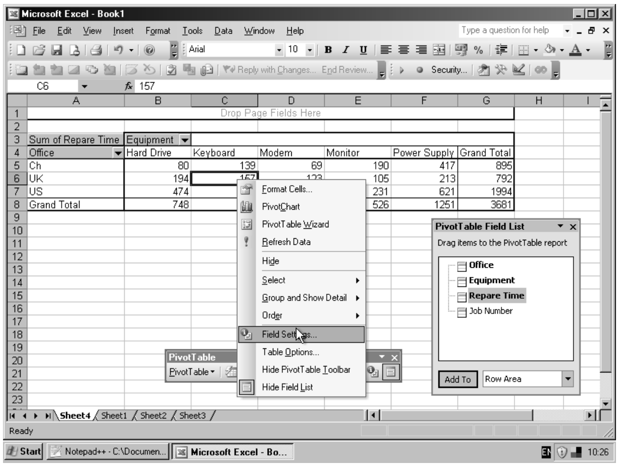
Figure 38: This screen shot shows the Pivot Table set up to summarise by office in Rows and equipment in Columns. The next step will be to tell Excel what data to actually summarise.



Sum of Repare Time	Equipment					
Office	Hard Drive	Keyboard	Modem	Monitor	Power Supply	Grand Total
Ch	80	139	69	190	417	895
UK	194	157	123	105	213	792
US	474	307	361	231	621	1994
Grand Total	748	603	553	526	1251	3681

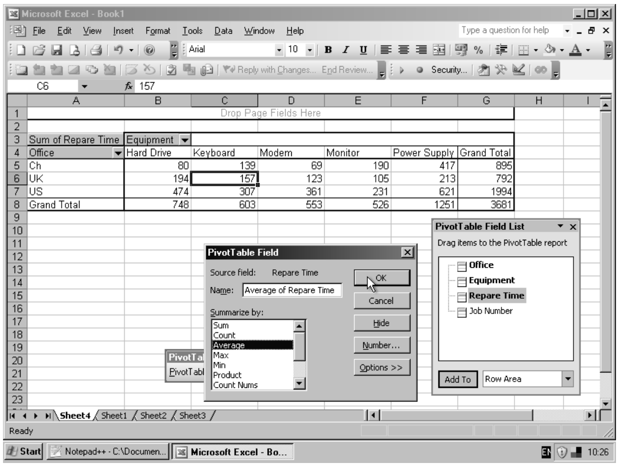
Figure 39: Dragging the "Repair Time" field from the Pivot Table Field List dialogue to the Data area causes the table to automatically summarise the sum of repair times for all offices and equipment.

Dragging a field into the Data area tells Excel that it is the field with the data to be summarised. Excel guesses the method of summary. In this case, Excel guesses that it should sum the values of the repair time field. For example, it takes all the repair times for hard drives in China and adds them together. In our screen shot above (figure 39) this gives 80 hours. This is not actually much use to us. We can change the method of summary as is shown in the next screen shot. For this example, I have chosen to summarise by showing the average (numeric mean) of the repair time.



Sum of Repare Time	Equipment					
Office	Hard Drive	Keyboard	Modem	Monitor	Power Supply	Grand Total
Ch	80	139	69	190	417	895
UK	194	157	123	105	213	792
US	474			231	621	1994
Grand Total	748			526	1251	3681

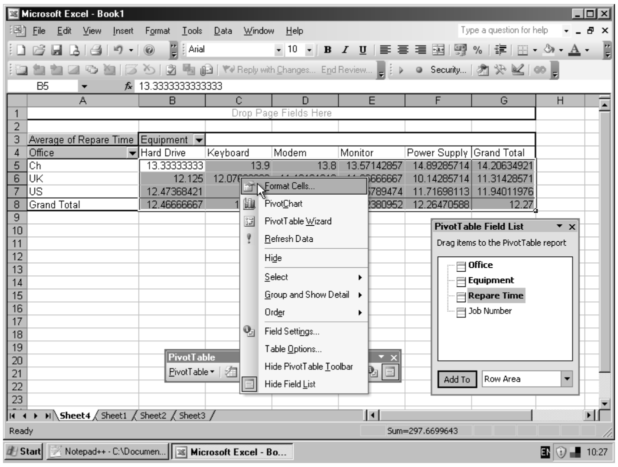
Figure 40: By right clicking one of the Cells in the Data area we get the context menu show above. By clicking the “Field Settings” option we get the “Pivot Table Field” dialogue as shown in the next screen shot.



Sum of Repare Time	Equipment					
Office	Hard Drive	Keyboard	Modem	Monitor	Power Supply	Grand Total
Ch	80	139	69	190	417	895
UK	194	157	123	105	213	792
US	474	307	361	231	621	1994
Grand Total	748	603	553	526	1251	3681

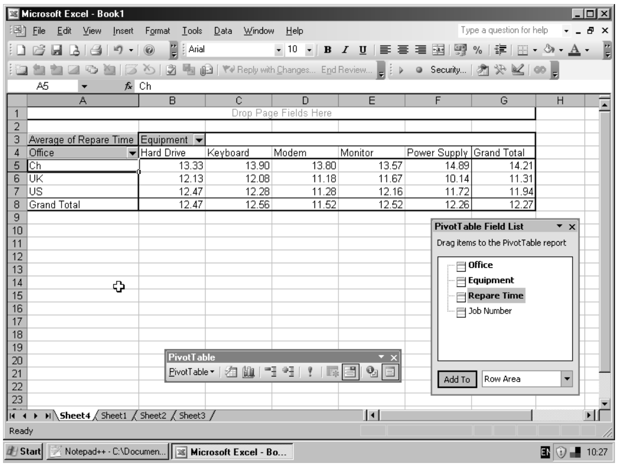
Source field: Repare Time
Name: Average of Repare Time
Summarize by: Average

Figure 41:The “Pivot Table Field” dialogue lets us change the way the data is summarised.



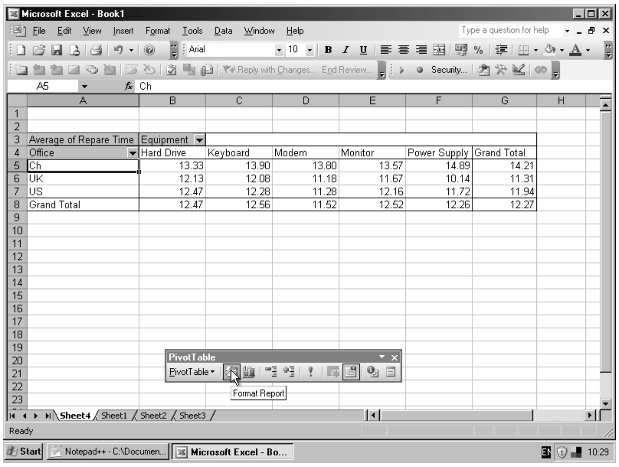
Average of Repare Time	Equipment					
Office	Hard Drive	Keyboard	Modem	Monitor	Power Supply	Grand Total
Ch	13.33333333	13.9	13.8	13.57142857	14.89285714	14.20634921
UK	12.125	[illegible]	[illegible]	[illegible]	10.14285714	11.31428571
US	12.47368421	[illegible]	[illegible]	[illegible]	11.71698113	11.94011976
Grand Total	12.46666667	[illegible]	[illegible]	[illegible]	12.26470588	12.27

Figure 42: In this example, when I had changed the summary method to average I got these very long decimal numbers. To make this less ugly, I used the normal right-click the "Format Cells" method.



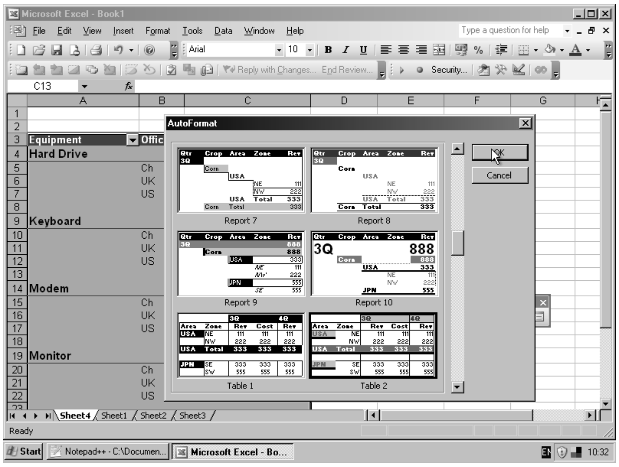
Average of Repare Time	Equipment					
Office	Hard Drive	Keyboard	Modem	Monitor	Power Supply	Grand Total
Ch	13.33	13.90	13.80	13.57	14.89	14.21
UK	12.13	12.08	11.18	11.67	10.14	11.31
US	12.47	12.28	11.28	12.16	11.72	11.94
Grand Total	12.47	12.56	11.52	12.52	12.26	12.27

Figure 43: Here it is, our Pivot Table in its default layout. For a lot of applications, this may well be sufficient. Should it be necessary to beautify things further, then Cell backgrounds, number formats and all these other things can be set by hand. However, doing so can make scripting a bit tricky; it might be better to initially see if one of the built in formats might do what we want.



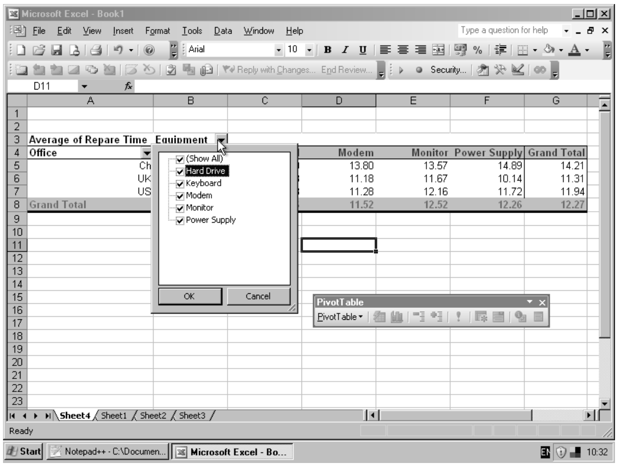
Average of Repare Time	Equipment					
Office	Hard Drive	Keyboard	Modem	Monitor	Power Supply	Grand Total
Ch	13.33	13.90	13.80	13.57	14.89	14.21
UK	12.13	12.08	11.18	11.67	10.14	11.31
US	12.47	12.28	11.28	12.16	11.72	11.94
Grand Total	12.47	12.56	11.52	12.52	12.26	12.27

Figure 44: Clicking the "Format Report" button on the Pivot Table tool bar reveals the "Auto Format" dialogue.



Equipment | Offic
Hard Drive: Ch, UK, US
Keyboard: Ch, UK, US
Modem: Ch, UK, US
Monitor: Ch, UK, US

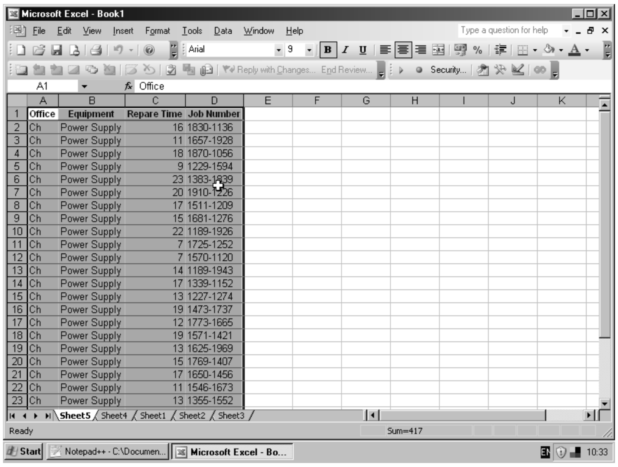
Figure 45: There are quite a few built in formats to choose from. It is worth being a bit careful however, several re-layout the Pivot Table field positions, which can be annoying. A click of the "OK" button applies a format to the actual table.



Average of Repare Time Equipment

Office	Modem	Monitor	Power Supply	Grand Total
Ch	13.80	13.57	14.89	14.21
UK	11.18	11.67	10.14	11.31
US	11.28	12.16	11.72	11.94
Grand Total	11.52	12.52	12.26	12.27

Figure 46: One of the things I like very much about Pivot Tables is the number of built in features. In this screen shot I have clicked the "Equipment" field tab. This then has let me choose which of the various values of the Equipment field I actually want displayed in the table. Not only is this a cool feature for me, it is also available to whomever I give my finished table.



Office	Equipment	Repare Time	Job Number
Ch	Power Supply	16	1830-1136
Ch	Power Supply	11	1657-1928
Ch	Power Supply	18	1870-1056
Ch	Power Supply	9	1229-1594
Ch	Power Supply	23	1383-1839
Ch	Power Supply	20	1910-1226
Ch	Power Supply	17	1511-1209
Ch	Power Supply	15	1681-1276
Ch	Power Supply	22	1189-1926
Ch	Power Supply	7	1725-1252
Ch	Power Supply	7	1570-1120
Ch	Power Supply	14	1189-1943
Ch	Power Supply	17	1339-1152
Ch	Power Supply	13	1227-1274
Ch	Power Supply	19	1473-1737
Ch	Power Supply	12	1773-1665
Ch	Power Supply	19	1571-1421
Ch	Power Supply	13	1625-1969
Ch	Power Supply	15	1769-1407
Ch	Power Supply	17	1650-1456
Ch	Power Supply	11	1546-1673
Ch	Power Supply	13	1355-1552

Figure 47: Another mind blowingly useful feature of a Pivot Table is drill-down. To get the newly created Sheet shown above, I double clicked on the "Ch/Power Supply" Data area Cell. This caused Excel to populate a new Sheet with just that data which as summarised to create that Cell.

With a Pivot Table we can use the summarised data to find the information we are interested in and drill-down to find the raw data that was summarised to create it. We should note that even though “Job Number” is not displayed in the Pivot Table, it is shown in the drill-down data.

If, for example, I felt that the “Ch/Power Supply” average was too high, I could:

1) Drill down to the raw data.
2) Sort by repair time.
3) Make a note of the top ten Job Numbers.
4) Call the appropriate manager and discuss what might be done to fix whatever made these jobs take so long.

Creating Pivot Tables Using Scripts

Looking At The Recorded Macro

The next block of code is the raw Recorded Macro as Excel 2003 created it from same session from which I took the above screen shots. At first blush it looks a bit scary (or at least it does not me). However, taken step by step, the beast can be tamed. So, my approach here will be to do just that, look at the script in little steps figuring out what is happening as we go along.

```
  ' Raw Recorded Macro

  ActiveWorkbook.PivotCaches.Add(SourceType:=xlDatabase, 
SourceData:= _
    "Sheet1!R1C1:R301C4").CreatePivotTable 
TableDestination:="", TableName:= _
    "PivotTable1", DefaultVersion:=xlPivotTableVersion10

  ActiveSheet.PivotTableWizard 
TableDestination:=ActiveSheet.Cells(3, 1)

  ActiveSheet.Cells(3, 1).Select

  ActiveWorkbook.ShowPivotTableFieldList = True
  With 
ActiveSheet.PivotTables("PivotTable1").PivotFields("Offi
ce")
    .Orientation = xlRowField
    .Position = 1
  End With
  With 
ActiveSheet.PivotTables("PivotTable1").PivotFields("Equi
pment")
    .Orientation = xlColumnField
    .Position = 1
  End With

  ActiveSheet.PivotTables("PivotTable1").AddDataField
ActiveSheet.PivotTables( _
    "PivotTable1").PivotFields("Repair Time"), "Sum of
Repair Time", xlSum

  Range("B5").Select

  ActiveSheet.PivotTables("PivotTable1").PivotFields("Su
m of Repair Time"). _
    Function = xlAverage

  Range("B5:G8").Select
  Selection.NumberFormat = "0.00"

  ActiveSheet.PivotTables("PivotTable1").PivotSelect "",
xlDataAndLabel, True

  ActiveSheet.PivotTables("PivotTable1").Format xlTable2
```

OK – brace ourselves – we're going in:

The whole process starts by setting up a Pivot Cache. Pivot Tables are a view of Data, the actual calculations that create that Data (i.e. all the summary calculations) are done in a special Object from the Class PivotCache. Each Excel Workbook has a Collection of PivtoCache Objects just like it has a Collection of Sheet Objects. To create a new PivotCache, we call the Add Method of the PivotCaches Collection of a Workbook Object.

```
  ActiveWorkbook.PivotCaches.Add(SourceType:=xlDatabase,
SourceData:= _
    "Sheet1!R1C1:R301C4").CreatePivotTable
TableDestination:="", TableName:= _
    "PivotTable1", DefaultVersion:=xlPivotTableVersion10
```

What is this all about? Here is a step by step guide to what Excel is doing here:

1) Take the PivotCaches Collection of the ActiveWorkbook (that which we are looking at) and Add a new PivotCache.

2) Set the data source type for this PivotCache to "xlDatabase".

3) Set the Range of Cells for this PivotCache to "Sheet1! R1C1:R301:C4".

4) Now that we have set up the PivotCache, create a Pivot Table from it.
5) Make the new Pivot Table's destination be “”. This makes it appear in a new Sheet.
6) Give the new Pivot Table the name “PivotTable1”.
7) Finally, give the new Pivot Table a default layout of “xlPivotTableVersion10”.

So in effect, these lines create a PivotCache Object which works with the Range of Cells which contains the raw data. It then creates a PivotTable Object which works with the the PivotCache to create the data summary we want.

```
  ActiveSheet.PivotTableWizard
TableDestination:=ActiveSheet.Cells(3, 1)

  ActiveSheet.Cells(3, 1).Select
  ActiveWorkbook.ShowPivotTableFieldList = True
```

This actually does nothing of use, it can be ignored!

```
  With
ActiveSheet.PivotTables("PivotTable1").PivotFields("Offi
ce")
    .Orientation = xlRowField
    .Position = 1
  End With
  With
```

This code uses the PivotTables Collection of the ActiveSheet (that Sheet which we are currently looking at) to locate our PivotTable. The fields which go into creating a PivotTable are called PivotFields. Each PivotField has a special Object representing it. These PivotField Objects can be found in the PivotFields Collection of a PivotTable Object. This is what the code does to find the “Office” field and make it a Row field.

ActiveSheet.PivotTables("PivotTable1").PivotFields("Equipment")

```
    .Orientation = xlColumnField
    .Position = 1
  End With
```

This piece of code is just the same as the previous one. The only difference is that it make the “Equipment” field into a Column.

```
    ActiveSheet.PivotTables("PivotTable1").AddDataField
ActiveSheet.PivotTables( _
        "PivotTable1").PivotFields("Repair Time"), "Sum of
Repair Time", xlSum
```

This piece of code uses the same method again to find the PivotTable Object. Once it has done so, it calls the AddDataField Method to add the "Repair Time" field as a Data field with a title of "Sum of Repair Time" and a summary function of xlSum.

```
    Range("B5").Select
```

This does nothing important however, the next bit does:

```
    ActiveSheet.PivotTables("PivotTable1").PivotFields("Su
m of Repair Time"). _
        Function = xlAverage
```

The above code converts the summary method from sum to average. Again, most of the actual code is made up of finding the PivotTable Object. When we translate over to a Scripting Macro, we will get rid of all this mess.

```
    Range("B5:G8").Select
    Selection.NumberFormat = "0.00"
```

The above is the bit that sets the numbers to just have two decimal places. I'll show an easier way of doing this later. The snag with this approach is that our script will have to work out what Range to use. It is better to use an approach which avoids this.

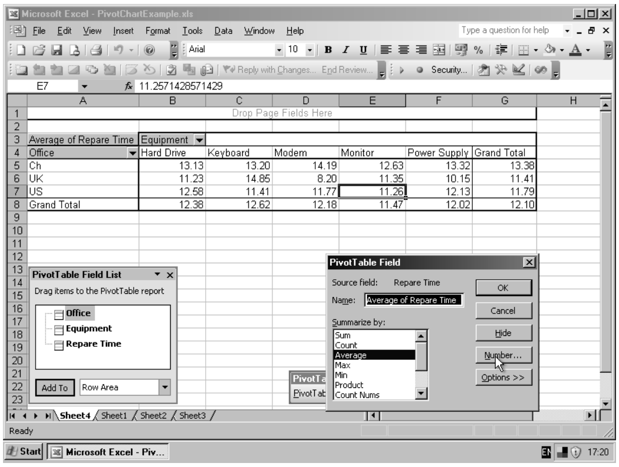
```
    ActiveSheet.PivotTables("PivotTable1").PivotSelect "",
xlDataAndLabel, True
```

This is not doing anything useful.

```
    ActiveSheet.PivotTables("PivotTable1").Format xlTable2
```

This final piece of code sets the format of the PivotTable to the built in format we wanted.

I did mention that there is a better way of setting the "Number Format" to two decimal places. The next two screen shots should how this is done from the graphic user interface. Whilst I was creating these screen shots, I had the Macro Recorder running, so we can see the code that came out as well.



Microsoft Excel - PivotChartExample.xls

E7 11.2571428571429

Drop Page Fields Here

Average of Repare Time	Equipment					
Office	Hard Drive	Keyboard	Modem	Monitor	Power Supply	Grand Total
Ch	13.13	13.20	14.19	12.63	13.32	13.38
UK	11.23	14.85	8.20	11.35	10.15	11.41
US	12.58	11.41	11.77	11.26	12.13	11.79
Grand Total	12.38	12.62	12.18	11.47	12.02	12.10

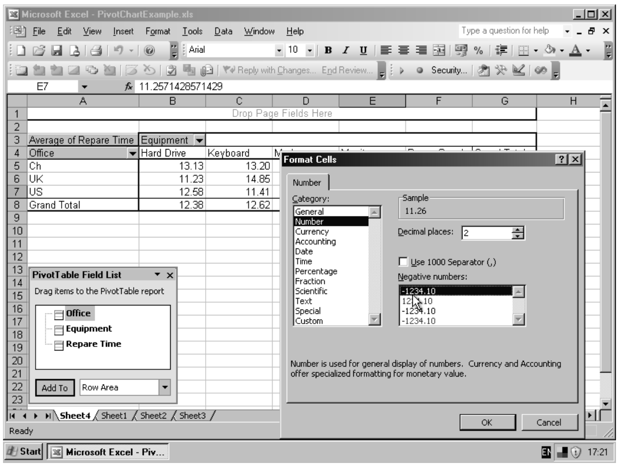
PivotTable Field

Source field: Repare Time

Name: Average of Repare Time

Summarize by: Average

Figure 48: We have already seen that right clicking in the Data area and selecting "Field Settings" causes the "PivotTable Field" dialogue to be shown. We can click the "Number" button on this dialogue to set the number format for the whole table.



Format Cells

Category: Number

Sample: 11.26

Decimal places: 2

Use 1000 Separator (,)

Negative numbers: -1234.10

Number is used for general display of numbers. Currency and Accounting offer specialized formatting for monetary value.

Figure 49: This is the result of clicking the "Number" button in the "PivotTable Field" dialogue. In this screen shot I am setting the whole Pivot Table to use a two decimal place format.

Using the "PivotTable Field" dialogue to set the number format of the PivotTable results in the following code:

```
 ' Fixing the number format without using a Range
 With
  ActiveSheet.PivotTables("PivotTable1").PivotFields(
_
```

```
        "Average of Repair Time")
        .NumberFormat = "0.00"
    End With
```

Here I have translated this code into scripting:

```
    ' Fixing number format translated
    With myTable.PivotFields("Average of Repair Time")
        .NumberFormat = "0.00"
    End With
```

Finished Product

Once we understand what each piece of the recorded Macro is doing, it is straight forward to translate into a Scripting Macro. Here I have placed all the translated Constants at the start of the script. I have also replaced all those clumsy lookup statements with Variables. In other words, each time it used to say something like:

```
ActiveSheet.PivotTables("PivotTable1").
```

I have just put:

```
myTable.
```

The finished product:

```
Option Explicit
    Dim  myWorkbook, myCache, myTable
    Const xlPivotTableVersion10 = 1
    Const xlRowField = 1
    Const xlColumnField = 2
    Const xlAverage = -4106
    Const xlTable2 = 11
    Const xlDatabase = 1


    ' Here we put code which creates an Excel.Appliation
    ' Object and  which gets the Workbook Object

    ' Here I split that massive statement in the recorded
    ' Macro into two steps; this is the first step to
    ' create the PivotCache. Also note that Excel has
    ' recorded named arguments (name:=value) we have to
    ' replace these with unnamed arguments (Appendix E).
    Set myCache = myWorkbook.PivotCaches.Add _
    ( _
      xlDatabase, _
```

```
        "Sheet1!R1C1:R301C4" _
    )
    ' This is the second step which creates the
    ' PivotTable from the PivotCache
    Set myTable= myCache.CreatePivotTable _
    ( _
        "", _
        "PivotTable1", _
        TRUE, _
        xlPivotTableVersion10 _
    )

    ' Now we have the table in existence, we can add
    ' the summary methods to it.
    With myTable.PivotFields("Office")
        .Orientation = xlRowField
        .Position = 1
    End With

    With myTable.PivotFields("Equipment")
        .Orientation = xlColumnField
        .Position = 1
    End With

    ' There is no point in a script of adding the
    ' Data field summarised as sum and then changing
    ' over to average.  Here I have added it as average
    ' to start with.
    myTable.AddDataField myTable.PivotFields("Repair
Time"), "Average of Repair Time", xlAverage

    ' Here is the nice clean number format approach
    myTable.PivotFields("Average of Repair Time")
            .NumberFormat = "0.00"
    End With

    ' Next we can set the auto-format we want
    myTable.Format xlTable2

    ' Finally, we can clean up a bit by getting rid
    ' of the field list and the  PivotTable command bar
    myExcel.CommandBars("PivotTable").Visible = False
    myWorkbook.ShowPivotTableFieldList = False
```

Tricks To Improve Data To Help Pivoting

Banding Data

Back in the definition of our test data, there is a section "*there is a new company wide directive that repairs should be done in fourteen hours or less.*" At first look, there is nothing in the raw data to help us here. Because there is nothing in the raw data, we cannot put anything in the Pivot Table. So, Scripting Macros come to the rescue. We can use a script to add a new Column which the appropriate data.

The following piece of script puts in a new Column. The value of this Column is 1 if a repair time was over 14 hours and 0 if it was not.

```
  mySheet.Cells(1,5).Value = "Over Limit"
  mySheet.Cells(2,5).Formula = "=if(C2>14,1,0)"
  mySheet.Range("E2:E301").FillDown
```

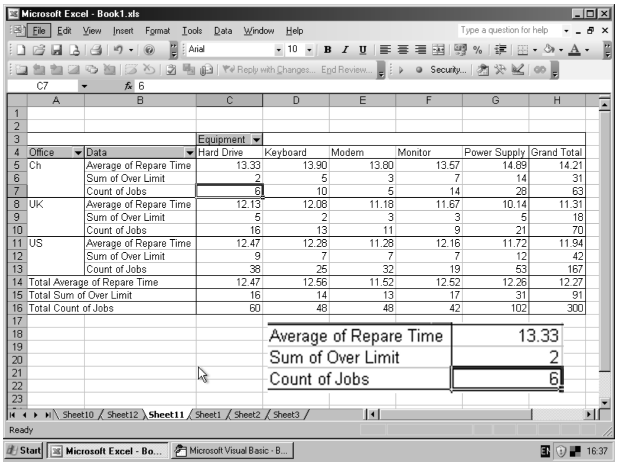
I have used a formula and the "FillDown" because this is a very fast approach. We could make the script inspect each value in Column C and put an appropriate value in Column E. The result would be identical, but it would run hundreds of times slower.

Now that we have a Column indicating if a datum is over time or not, we can add these new data into the Pivot Table. I have added the "Over Limit" field to the Data area twice. The first time I have added it with a summary function of "Sum". This results in a number showing how many repairs were over the limit for a particular combination of Office and Equipment. The second time I have added it with a summary function of "Count". This simply gives the total number of repair for a particular combination of Office and Equipment.

I call this approach banding. The idea is that we take data which is continuous (could be anything out of a range of values) and sort it into two or more bands. In this example, we make two bands, over 14 hours and not over 14 hours. Another example might be people's heights. With heights we might make a less than 1.5 meters, 1.5 to 2 meters and greater than 2 meters.

Here is the code which creates the banding and the Pivot Table for our repair time example:

```
  Option Explicit
  ' ... Some other code goes here ...

  Dim myTable,myCache
  Const xlDatabase = 1
  Const xlRowField = 1
  Const xlColumnField = 2
  Const xlPivotTableVersion10 = 1
  Const xlSum = -4157
  Const xlCount = -4112
  Const xlAverage = -4106

  mySheet.Cells(1,5).Value = "Over Limit"
  mySheet.Cells(2,5).Formula = "=if(C2>14,1,0)"
  mySheet.Range("E2:E301").FillDown

  Set myCache = myWorkbook.PivotCaches.Add _
  ( _
    xlDatabase, _
    "Sheet1!R1C1:R301C5" _
  )

  Set myTable=myCache.CreatePivotTable _
  ( _
    "", _
    "PivotTable1", _
    true, _
    xlPivotTableVersion10 _
  )

  With myTable.PivotFields("Office")
    .Orientation = xlRowField
    .Position = 1
  End With

  With myTable.PivotFields("Equipment")
    .Orientation = xlColumnField
    .Position = 1
  End With

  myTable.AddDataField _
  myTable.PivotFields("Repare Time"), _
    "Average of Repare Time",_
    xlAverage

  myTable.AddDataField _
    myTable.PivotFields("Over Limit"), _
    "Sum of Over Limit", _
    xlSum

  myTable.AddDataField _
    myTable.PivotFields("Over Limit"), _
    "Count Of Jobs", _
    xlCount
```



		Equipment					
Office	Data	Hard Drive	Keyboard	Modem	Monitor	Power Supply	Grand Total
Ch	Average of Repare Time	13.33	13.90	13.80	13.57	14.89	14.21
	Sum of Over Limit	2	5	3	7	14	31
	Count of Jobs	6	10	5	14	28	63
UK	Average of Repare Time	12.13	12.08	11.18	11.67	10.14	11.31
	Sum of Over Limit	5	2	3	3	5	18
	Count of Jobs	16	13	11	9	21	70
US	Average of Repare Time	12.47	12.28	11.28	12.16	11.72	11.94
	Sum of Over Limit	9	7	7	7	12	42
	Count of Jobs	38	25	32	19	53	167
Total Average of Repare Time		12.47	12.56	11.52	12.52	12.26	12.27
Total Sum of Over Limit		16	14	13	17	31	91
Total Count of Jobs		60	48	48	42	102	300

Average of Repare Time	13.33
Sum of Over Limit	2
Count of Jobs	6

Figure 50: Here we can see the Pivot Table with banded data. I have included an enlarged view of the "Ch/Hard Drive" summary data. We can see that there were 6 jobs in total of which 2 were over the 14 hour repair limit. However, the average repair time is 13.33 hours, which might be considered a bit close to the limit for comfort. Remember though, there data are totally invented!

Cleaning Data

In Lesson 5 we saw that real world data is 'dirty'. Pivot Tables are particularly susceptible to this problem. If we consider our repair time example; what would happen if in the data there is a Row which has 'ch' rather than 'Ch' as the Office code? This would actually result in a whole new Row in the Pivot Table. Both the data in the new Row and in the old 'Ch' Row would be incorrect because they should all be summarised into the same Row.

Simply put, cleaning up real world data is the differentiator between a well done Pivot Table Macro Script and a misleading mess.

The Page Area

Back when we looked at placing fields into different parts of a blank Pivot Table I mentioned the Page area. Fields placed here make a 'filter' for the entire table. If, for example, we had data on dogs and we wanted to see a summary for just one breed at a time, then we would put the field "Breed" into the Page area.

Lesson 9: Working 'Outside The Box'

Accessing Binary Files

So far in Baby Steps we have looked at reading and writing text files. Sometimes we might want to access binary files. An example reason for doing this might be to read the comment in a mp3 file.

The key to accessing binary files from scripts is to use the ADODB library which is supplied by Microsoft for free on all Windows distributions. You will require ADODB 2.5 or later. If you want to know if you have this versio then the following script will tell you:

```
dim adodb
set adodb=WScript.CreateObject("ADODB.Connection")
dim v
WScript.echo "Your ADODB Version=" & adodb.version
```

Any thing at 2.5 or later should work fine.

```
' This is a simple example of managing binary files in
' VBScript using the ADODB object
Dim inStream,outStream
Const adTypeText=2
Const adTypeBinary=1

' We can create the scream object directly, it does
' not need to be built from a record set or anything
' like that
Set inStream=WScript.CreateObject("ADODB.Stream")

' We call open on the stream with no arguments.
' This makes the stream become an empty container
' which can be then used for what operations we want

inStream.Open
inStream.type=adTypeBinary

' Now we load a really BIG file.  This should show
' if the object reads the whole file at once or just
' attaches to the file on disk
' You must change this path to something on your
' computer!
inStream.LoadFromFile "C:\temp\test.gif”

' Copy the data over to a stream for outputting
set outStream=WScript.CreateObject("ADODB.Stream")
outStream.Open
outStream.type=adTypeBinary
```

```
  dim buff
  buff=inStream.Read()

  ' Write out to a file
  outStream.Write(buff)

  ' You must change this path to something on your
  ' computer!
  outStream.SaveToFile  "C:\temp\test.gif”

  outStream.Close
  inStream.Close
```

In the above example the Stream object is used to load the contents of the file into memory and store it as a Byte array (which is a special way of storing data in memory which VBScript does not normally use). The byte array is then passed to another stream which is written out to disk. Not very memory efficient, but if we want really efficient binary file handling, then you should probably be programming in C! this technique comes in handy as a quick and easy approach which does not require any new compiled executables.

Another way of achieving the same thing is to use the CopyTo method. This does not allow you access to any of the data, but is more efficient if you are wanting to just copy a file. But then, if all we want to do is copy a file, why not use the Scripting.FileSystemObject CopyFile method?

One problem you might face using this approach in VBScript is that we cannot create a Byte array. The following script fails:

```
  dim outStream
  const adTypeText=2
  const adTypeBinary=1
  dim data

  ' Copy the dat over to a stream for outputting
  set outStream=WScript.CreateObject("ADODB.Stream")
  outStream.Open
  outStream.type=adTypeBinary

  dim buff()
  redim buff(255)
  dim i
  for i=0 to 255
    buff(i)=i
  next

  ' Write out to a file - but the script fails here
  outStream.Write(buff)
  outStream.SaveToFile  "C:\temp\test.gif”

  outStream.Close
  inStream.Close
```

The script fails because the array created is a Variant array, not a Byte array. What does that mean? Well here goes.

VBScript tries to make life for the programmer simple. One way it does this is to say that any Variable can hold any type of data. This is actually a sort of trick that works in the background. Variables in VBScript are stored as a thing called a '*Variant*'. A Variant holds not only a value, but information on type. So a Variable holding a String in VBScript not only holds the data for the String its self, but an extra datum which in effect says “This is a String”.

Other pieces of software, ADODB being an example, do not use Variants. VBScript can translate for them. So ADODB.Stream will pass data in a from VBScript as an Array of Bytes. VBScript adds to the Variable the datum saying “This is an Array of Bytes” when it reads from the ADODB.Stream. When VBScript writes to the ADODB.Stream it strips this extra datum off again.

All this 'magic' works just fine until to comes to creating a Variable which holds a Array to be written to ADODB.Stream. VBScript cannot create and Array of Bytes. It can only create an Array of Variants. The magic does not automatically convert an Array of Variants to and Array of Bytes. To work around this problem, we can get ADODB.Stream to create an Array of Bytes for us by writing out a text file of the appropriate length and then reading it in. I gave this a lot of thought and decided that writing up this method would be just far too hard core for 'Baby Steps'. However, I will be happy to answer any emails on on the subject and might write it up in my blog.

Reading Data From Binary Files – An Example

OK, here we see that we can read binary files as long as they are not too large. We can read the data in them by getting a Byte array from the LoadFromFile method of the Stream object, and we can write to them. For the last piece of this section, let us do something 'real' with this technology. The script below looks for all the '.gif' files in 'My Pictures' and lists them along with the contents of the GIF version label. The GIF version label is stored at the start of the 'gif' binary files.

```
  Option Explicit
  Dim myShell, sendToFolder, myPicturesPath, myShortcut
  Dim fso,myFolder, myFile, fileName, comment, myExcel
  Dim myWorkbook, myRow, mySheet
  ' Find "My Pictures"
  Set myShell = CreateObject("WScript.Shell")
  myPicturesPath = myShell.SpecialFolders("MyDocuments")
& "\My Pictures"

  ' Open "My Pictures" as a folder so we can see
  ' which files are inside it
  Set fso=CreateObject("Scripting.FileSystemObject")
  Set myFolder=fso.GetFolder(myPicturesPath)

  ' Set Up Excel To receive The Data
  Set myExcel=CreateObject("Excel.Application")
  Set myWorkbook=myExcel.Workbooks.Add
  Set mySheet=myWorkbook.Sheets(1)
  myRow=2
  mySheet.Cells(1,1).Value="Name"
  mySheet.Cells(1,2).Value="GIF Type"
  myExcel.Visible=TRUE

  ' Loop through each file found and see
  ' if its file extension is .gif
  ' If a file is a .gif file then call our function
  ' which opens it as a binary file and reads the
  ' version label
  for each myFile in myFolder.Files
    fileName=myFile.name
      fileName=Lcase(fileName)
      if Right(fileName,4)=".gif" then
        ' Read the version label
        comment=GetGifComment(myFile.path)
        ' Place the data in the spreadsheet
        mySheet.Cells(myRow,1).Value=fileName
        mySheet.Cells(myRow,2).Value=comment
        ' Step down to the next Row
        myRow=myRow+1
      end if
  next

  ' Make the spreadsheet look a bit nicer
  With mySheet.Range("A1:B1").Font
    .FontStyle = "Bold"
    .Size = 12
  End With

  mySheet.Columns(1).Autofit
  mySheet.Columns(2).Autofit
  'Script ends here

  function GetGifComment(gifFilePath)
    dim inStream,buff,commentLen,commentStr,myIndex
    dim myByte,myByteValue,myCharacter

    set inStream=WScript.CreateObject("ADODB.Stream")

    inStream.Open
    inStream.type=1

    inStream.LoadFromFile gifFilePath

    buff=inStream.Read()
    inStream.Close

    commentStr=""
    for myIndex = 1 to 6
      ' Extract 1 byte from the buffer
      myByte = MidB(buff,myIndex,1)
      ' Gets its numeric value
      myByteValue = AscB(myByte)
      ' Convert that numeric value into a character
      myCharacter  = Chr(myByteValue)
      ' Append that character to the string
      commentStr  = commentStr & myCharacter
    next
    GetGifComment = commentStr
  end function
```

ADODB.Stream Summary

Properties

Property	Description
CharSet	Sets or returns a value that specifies into which character set the contents are to be translated. This property is only used with text Stream objects (type is adTypeText)
EOS	Returns whether the current position is at the end of the stream or not
LineSeparator	Sets or returns the line separator character used in a text Stream object
Mode	Sets or returns the available permissions for modifying data
Position	Sets or returns the current position

	(in bytes) from the beginning of a Stream object
Size	Returns the size of an open Stream object
State	Returns a value describing if the Stream object is open or closed
Type	Sets or returns the type of data in a Stream object

Methods

Method	Description
Cancel	Cancels an execution of an Open call on a Stream object
Close	Closes a Stream object
CopyTo	Copies a specified number of characters/bytes from one Stream object into another Stream object - args Stream,[count]
Flush	Sends the contents of the Stream buffer to the associated underlying object
LoadFromFile	Loads the contents of a file into a Stream object
Open	Opens a Stream object
Read	Reads the entire stream or a specified number of bytes from a binary Stream object
ReadText	Reads the entire stream, a line, or a specified number of characters from a text Stream object
SaveToFile	Saves the binary contents of a Stream object to a file
SetEOS	Sets the current position to be the end of the stream (EOS)
SkipLine	Skips a line when reading a text Stream
Write	Writes binary data to a binary Stream object
WriteText	Writes character data to a text Stream object

If the mode of a stream is adTypeText (2) we will get an error if you try to use Read or Write. Equally, if the type is adTypeBinary (1) the ReadText and WriteText will fail. Note also that if the streams are set to Text and CopyTo is used to copy a file, the file will be converted to text even if it did not start out this way. The end result is normally a corrupt file. I would strongly suggest always using adTypeBinary with ADODB and using the FileSystemObject for all text file manipulations.

Sending Key Strokes To Other Applications Through

So far we have seen the ability of scripts to work with Excel using COM. How about using a script to work with Excel and another application? An example application might be Notepad. However, there is no COM interface to Notepad, so we are going to have to use the WScript.SendKeys method. So here is a quick crib on how that works.

Most keys can be represented by the character of the key itself. e.g, the key sequence FRED can be represented simply by "FRED". Here is a simple script which will open Notepad and type FRED into it. To do that, it creates a Object from the Wscript.Shell Class. This Object has a Method to 'run' an application. This Method is used to run Notepad. Once Notepad is up and running, the key sequence F·R·E·D is sent to it.

```
Option Explicit
Dim WshShell

' Open notepad
Set WshShell = CreateObject("WScript.Shell")
WshShell.Run "notepad", 9

' Give Notepad time to load (500 milliseconds)
WScript.Sleep 500

' Write FRED
WshShell.SendKeys "FRED"
```

Some special keys, such as the control keys, function keys etc are encoded in a string enclosed by {braces}

Key	Code
{	{{}
}	{}}
[	{[}
]	{]}
~	{~}
+	{+}
^	{^}
%	{%}

Key	Code
BACKSPACE	{BACKSPACE}, {BS}, or {BKSP}
BREAK	{BREAK}
CLEAR	{CLEAR}
CAPS LOCK	{CAPSLOCK}
DEL or DELETE	{DELETE} or {DEL}
DOWN ARROW	{DOWN}
END	{END}
ENTER	{ENTER} or ~
ESC	{ESC}
HELP	{HELP}
HOME	{HOME}
INS or INSERT	{INSERT} or {INS}
LEFT ARROW	{LEFT}
NUM LOCK	{NUMLOCK}
PAGE DOWN	{PGDN}
PAGE UP	{PGUP}
PRINT SCREEN	{PRTSC}
RIGHT ARROW	{RIGHT}
SCROLL LOCK	{SCROLLLOCK}
TAB	{TAB}
UP ARROW	{UP}
F1	{F1}
F2	{F2}
F3	{F3}
F4	{F4}
F5	{F5}
F6	{F6}
F7	{F7}
F8	{F8}
F9	{F9}
F10	{F10}
F11	{F11}
F12	{F12}
F13	{F13}
F14	{F14}
F15	{F15}
F16	{F16}

To specify keys combined with any combination of SHIFT, CTRL, and ALT keys, precede the key code with one or more of the following:

```
For SHIFT prefix with +
For CTRL  prefix with ^
For ALT   prefix with %
```

Here is a fuller example in which we create a small table in and Excel spreadsheet and use copy and paste to place the contents into Notepad.

```
Option Explicit
Dim WshShell, myExcel, myWorkbook, mySheet

' Open Excel
Set myExcel = CreateObject("Excel.Application")
' Create a Workbook and set myWorkbook to refer to it
' all in one go
Set myWorkbook = myExcel.Workbooks.Add()
Set mySheet = myWorkbook.Sheets(1)
mySheet.Cells(1,1).Value="Hellow"
mySheet.Cells(1,2).Value="World"
mySheet.Cells(1,3).Value="This"
mySheet.Cells(2,1).Value="Is"
mySheet.Cells(2,2).Value="An"
mySheet.Cells(2,3).Value="Example"
mySheet.Cells.Copy

' Tell Excel not to ask if the document
' should be saved or to show any other alerts
myExcel.DisplayAlerts = FALSE

' Shut down Excel as we do not require it any more
myExcel.Quit

' Open notepad
Set WshShell = CreateObject("WScript.Shell")
WshShell.Run "notepad", 9

WScript.Sleep 500

' Do The Paste
WshShell.SendKeys "%e"
WshShell.SendKeys "p"
```

Stopping Excel From Complaining

In the above example of sending key strokes to Notepad, I used a trick to stop Excel complaining about closing without saving. Sometimes when we are processing data, we want to use an instance of Excel but not bother saving the changes we make to it. The way to do this is by the Application.DisplayAlerts Property.

Choice And Decision Processing

Yes – Excel Really Is This Powerful!

It is amazing but true, Excel can perform the sort of complex decision making tasks we more normally associate with software systems which cost millions. As a software architect, I have often been asked to integrate choice/decision processing into software systems. Achieving this is usually daunting in the extreme. However, with Excel the challenge falls away leaving a straight-forward step wise approach and minimal cost.

What Is Choice/Decision Processing?

Should Burt get a new car this year? Which air-conditioner units should be serviced and which replaced? Which invoices must be honoured this month and which can wait till next?

All of these examples are choice/decision processing challenges. In computer science it is called "logical programming". It turns out that people find it hard to be consistent with logical tasks like these. It also takes a lot of time for people to process data through logical sequences. Even when we layout careful rule sets, humans just struggle to apply them uniformly. In some ways this is the greatest asset of humans. We constantly review decisions and bring in external factors whilst considering them. Humans do not tend to blindly follow obviously incorrect rules. Computers just process data and rules over and over again without variation or consideration. For some situations the 'common sense' of humans is ideal; for others we need the uniformity, reliability and and speed of a computer.

Whilst we often think of Excel as a number crunching tool, it turns out to be just fantastic at processing logical problems as well. Excel by its self is brilliant at applying complex, or even very complex rule sets to a data to make choices. So, by its self, Excel is the prefect for working out if Burt gets his new car. Macro Scripting then adds the ability for Excel to munch through thousands of data sets making decisions for each.

With a simple script, and Excel, we can make logical decision processing systems which are able to make thousands of decisions in a very short period of time. This approach could allocate new cars, new computers or even new benefit packages to entire payrolls. Such abilities with "dedicated systems" cost thousands, if not millions, of Pounds, Euros or Dollars to implement; most of us already own a copy of Excel and can implement such a system for the price of dinner.

First Off – Just The Very Minimum Of Logic

Logic and logical programming are a library's worth of knowledge. We want to take a Baby Step. So, here we will look at the absolute minimum of logic we can get away with and still be one hundred percent capable of making Excel based decision systems.

TRUE & FALSE

In this approach, all questions have one of two possible answers: TRUE or FALSE.

We ask a question by saying "Is this TRUE or FALSE". For example, take a blank spreadsheet and put this formula into the top left hand corner:

```
=1>0
```

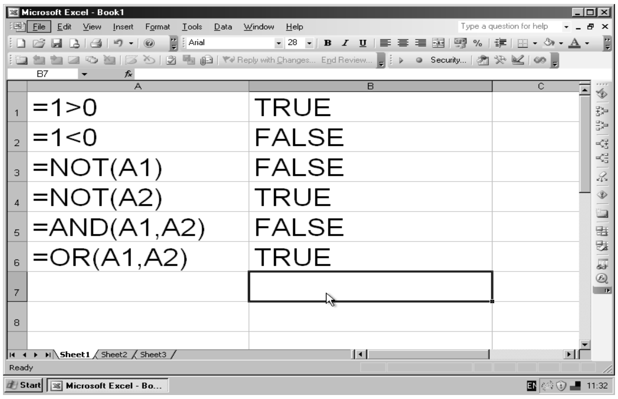
The Cell should show TRUE. What we have done is ask the question "Is 1 greater than 0". Excel has answered TRUE.

There are no prizes then for figuring out what ...

```
=1<0
```

... gives. Yes, it is FALSE.

We can substitute Cell references into these formulae. So if we place the number 1 in Cell A1 and =A1>0 into A2 then Excel will report the value of A2 as TRUE. Change A1 to 0 and A2 will change to FALSE. Figure 51 shows several such examples.



	A	B	C
1	=1>0	TRUE	
2	=1<0	FALSE	
3	=NOT(A1)	FALSE	
4	=NOT(A2)	TRUE	
5	=AND(A1,A2)	FALSE	
6	=OR(A1,A2)	TRUE	
7			
8			

Figure 51: This screen shot shows the concept of TRUE/FALSE *formulae in Excel. It then goes on to illustrate how to process the answers to these 'questios' using* AND, OR *and* NOT. *In the first column are the Excel formulae (displayed via Tools/Options/Formulas) and in the second column are the* TRUE/FALSE *results.*

Chaining Decisions To Make Rules

Now that we can get Excel to answer logical questions it would be nice to get it to chain the questions together. If we consider the rule:

> "You get a car if your car is older than 2 years old AND you are in the sales department"

I have capitalised 'AND' in the above rule because it does the joining work which chains two separate logical questions together.

"Is our care older than 2 years?

AND

Are you in sales?"

Consider, if you would, a spreadsheet like this:

	A	B
1	Age Of Car (Years)	3
2	Department	Sales

We can take these data and use them to create a decision:

	A	B	C
1	Age Of Car (Years)	3	
2	Department	Sales	
3	Choice	=B1>2	=AND(B3,B2="Sales")

In the above spreadsheet Cell C3 will display a value of TRUE. If we change B2 to be something other than 'Sales' or we change B1 to be 2 or less, then C3 will display a value of FALSE. In other words, Excel has chained two choices together to make a rule.

The key word AND() in an Excel formula means a special Excel Function. A comma separated list of *arguments* goes in the parenthesis. AND() is TRUE if and only if every single argument is TRUE.

```
=AND(TRUE,TRUE,FALSE)      → FALSE
=AND(TRUE,TRUE,TRUE)       → TRUE
=AND(TRUE,FALSE)           → FALSE
=AND(FALSE,TRUE)           → FALSE
=AND(TRUE,TRUE)            → TRUE
```
etc...

Sometimes is makes more sense to ask when something is not true. Consider that people do not get a car if they live in Holland (it is too flat to bother, one should use a bike). One way to look at this decision is:

> "A person may be eligible for a car if they do NOT live in Holland"

Excel has a function NOT() that does this for us:

```
=NOT(1>2) → TRUE
```

The above formula asks "Is it TRUE or FALSE that 1 is NOT greater than 2". Well, 1 is not greater than 2, so the answer is TRUE. Likewise, the following formula will give a value of FALSE:

```
=NOT(1>0) → FALSE
```

We can use our new tool to add the "no cars in Holland" decision to our rule processing spreadsheet.

	A	B	C	D
1	Age Of Car (Years)	3		
2	Department	Sales		
3	Country	France		
4	Choice	=B1>2	=AND(B3,B2="Sales")	=AND(C4, NOT(B3="Holland"))

In the example above, Burt will get a new car because the value of D4 will be TRUE. If he moved to Holland then he would not get the car. Excel can make that choice for us using this simple step by step rule system.

There are probably thousands of ways to load this decision system into Excel; using the way I have shown it is simple to create and simple to maintain the spreadsheet. The approach can be summed up like this:

- Place decisions in a Row.
- Make only one TRUE/FALSE decisions in each each Cell of that Row.
- Chain the Cells together so each Cell's decision is chained by AND to the result of the Cell to its left.

- The right most Cell in a Row gives the actual output choice TRUE/FALSE of the rule.

What we have done so far is build single decisions into rules using chaining. How about if there are more outcomes than just TRUE/FALSE or “Get a car / don't get a car”? What about making Excel pick what sort of car to give? This is where blocks and zeds come in.

Blocks And Zeds

Rules are groups of decisions chained together. We place all the decisions for a rule in one Row. Blocks are groups of rules which interact with each other. If the outcome of a rule has no impact on the outcome of another rule then the two rules are in different blocks. But if the outcome of a rule affects the outcome of another rule then the two rules are in the same block.

Consider the next set of decisions which go to make up two rules. The idea being that if an employee is in Sales but will not be getting a new car this year, then they get a 'bus time table instead.

	A	B	C	D
1	Age Of Car (Years)	3		
2	Department	Sales		
3	Country	France		
4	Get Car ?	=B1>2	=AND(B3,B2=“Sales”)	=AND(C4, NOT(B3=“Holland”))
5	Get 'Bus Time Table?	=NOT(D4)	=AND(B5,B2=“Sales”)	

Now Excel computes the answers to two questions:

1. Does Burt get a new car?
2. Does Burt get a 'bus time table?

Cell C5 depends on B5 which depends on D4 which depends on C4 which depends on B4. If we connect up these Cells in this order with a line (like join-the-dots) then we will draw a Z shape. Plus, the rules in Rows 4 and 5 make up a block because they depend on each other. So, these rules make a block and a zed.

Putting It All Together

Now we are going to look at a completely fresh example. In this example we are going to introduce the use of more than one block and a new function OR. OR is the last logical function we will need to look at.

```
=OR(TRUE,TRUE)                → TRUE
=OR(TRUE,FALSE)               → TRUE
=OR(FALSE,FALSE)              → FALSE
=OR(FALSE,FALSE,TRUE)         → TRUE
```

OR gives TRUE if any of its arguments are TRUE. It says “If my first argument OR my second argument OR my third argument ... are TRUE then I am TRUE”.

In decision/choice work with Excel, we do not use OR to chain decisions to make a Rule. We use OR to collect together answers from Rules. For example, we might want to say something like “If Rule 2 Or Rule 4 say Burt gets a car, then he gets a car”.

Here is the scenario for our next example:

There is a company which is intending to upgrade from MS Office 2003 to MS Office 2007.

However, there are some complexities. First off, not everyone is to be upgraded to 2007; the company has a deal by which some of the Office 2003 licenses can be recycled to allow upgrade of old Office 2000 license for people who do not require the latest software. Also, there are three types of Office 2007 license packages:

1. Word, Excel, Outlook, Power-Point and Access
2. Word, Excel, Outlook
3. Word, Excel

There are three departments to this company:

1. Admin
2. Sales
3. Supply

Admin require Office 2007 because some of the administration systems are being upgraded and require the this version of Office for integration.

So, here are the rules for choosing what software to install on people's machines:

- People in Admin who have Access version > 9 get license package 1 otherwise, license package 2.
- People with Power-Point get Power-Point again.
- Sales only require Word and Excel because they travel and will use WebMail for email.
- Supply get an upgrade to Office 2003 (Office 11) if they do not already have it. There is only one Office 2003 license package.

This example is run using three Sheets in one Workbook. The first is "DataIn". This Sheet is where the information about the employee is placed. The next Sheet is "logic". This is where all the rules are placed. "logic" takes its information from "DataIn". Finally, there is "DataOut". This Sheet is just a presentation report to help make the results unambiguous.

Here is the structure of the "DataIn" Sheet:

User Information	
Title	*Datum*
Name	Fred
Department	admin
Installed Software	
Package	*Version (0 means Not Installed)*
Word	11
Excel	11
Outlook	11
Powerpoint	1
Access	10

Here is the structure of the "Logic" sheet:

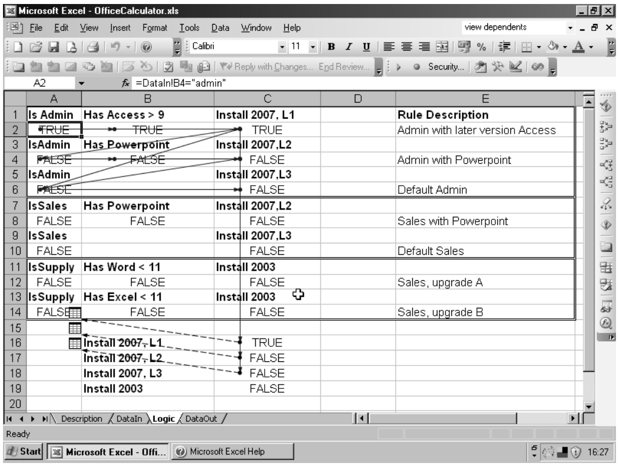
Is Admin	*Has Access > 9*	*Install L1*
=DataIn!.B4="admin"	=AND(A2,DataIn!.B12>9)	=B2
IsAdmin	*Has Powerpoint*	*Install L2*

=AND(NOT(C2), DataIn!.B4="admin")	=AND(A4,DataIn!.B11>0)	=B4
IsAdmin		*Install L3*
=AND(NOT(C4),NOT(C2),DataIn!.B4="admin")		=A6
IsSales	*Has Powerpoint*	*Install L2*
=DataIn!.B4="sales"	=AND(A8,DataIn!.B11>0)	=B8
IsSales		*Install L3*
=AND(NOT(C8),DataIn!.B4="sales")		=A10
IsSupply	*Has Word < 11*	*Install 2003*
=DataIn!.B4="supply"	=AND (A12,DataIn!.B8<11)	=B12
IsSupply	*Has Excel < 11*	*Install 2003*
=AND(NOT(C12);DataIn!.B4="supply")	=AND(A14;DataIn!.B9<11)	=B14

At the bottom of the "Logic" Sheet, OR functions are used to collect the output of the rules:

Install 2007, L1	=C2
Install 2007, L2	=OR(C4;C8)
Install 2007, L3	=OR(C6;C10)
Install 2003	=OR(C12;C14)

The nature of the rules means that a employee will get only one of the choices. In other words, only one of the install options will show TRUE.


Microsoft Excel - OfficeCalculator.xls
File Edit View Insert Format Tools Data Window Help
view dependents
Calibri
11
Reply with Changes... End Review...
Security...
Description DataIn Logic DataOut
Ready
Start
Microsoft Excel - Offi...
Microsoft Excel Help
16:27

A2 =DataIn!B4="admin"

	A	B	C	D	E
1	Is Admin	Has Access > 9	Install 2007, L1		Rule Description
2	TRUE	TRUE	TRUE		Admin with later version Access
3	IsAdmin	Has Powerpoint	Install 2007,L2		
4	FALSE	FALSE	FALSE		Admin with Powerpoint
5	IsAdmin		Install 2007,L3		
6	FALSE		FALSE		Default Admin
7	IsSales	Has Powerpoint	Install 2007,L2		
8	FALSE	FALSE	FALSE		Sales with Powerpoint
9	IsSales		Install 2007,L3		
10	FALSE		FALSE		Default Sales
11	IsSupply	Has Word < 11	Install 2003		
12	FALSE	FALSE	FALSE		Sales, upgrade A
13	IsSupply	Has Excel < 11	Install 2003		
14	FALSE	FALSE	FALSE		Sales, upgrade B
15					
16		Install 2007, L1	TRUE		
17		Install 2007, L2	FALSE		
18		Install 2007, L3	FALSE		
19		Install 2003	FALSE		
20					

Figure 52: Excel has the ability to draw out the dependencies between Cells. By doing this we can see the structure of the Blocks and Zeds in our Workbook. I have put dark borders around each block to make easier to see the structure of the Logic Sheet.

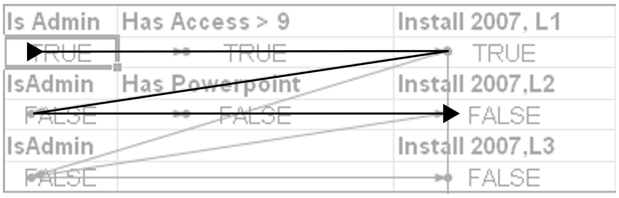


Figure 53: Working with Zeds! We load the choices into Excel by starting at the top left and working across to the right. Then we start again at the left. Each line in a group is linked to the answer of the line above so that only one line in a group can be TRUE. This means the dependencies in between the lines make a Z pattern.

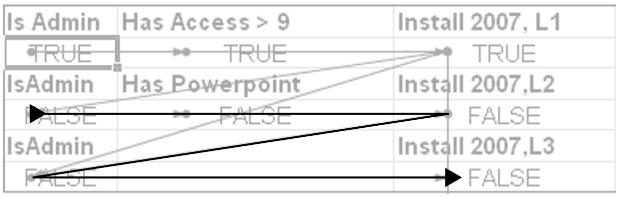
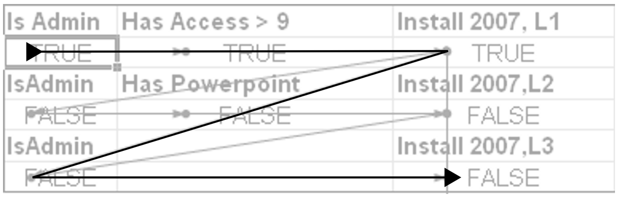
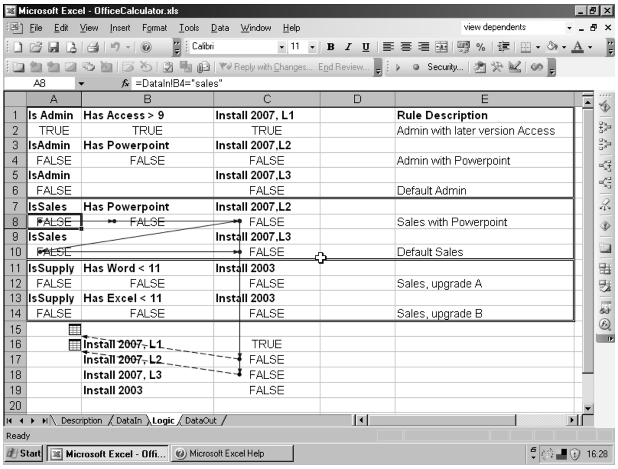


Figure 54: In the above group, there are three lines of choices yielding three possible outcomes. The bottom line makes a Z with the middle line.



Is Admin	Has Access > 9	Install 2007, L1
TRUE	TRUE	TRUE
IsAdmin	Has Powerpoint	Install 2007,L2
FALSE	FALSE	FALSE
IsAdmin		Install 2007,L3
FALSE		FALSE

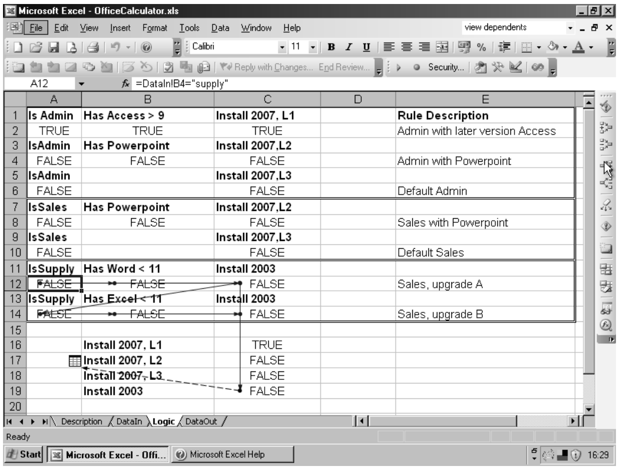
Figure 55: Here we can see the group with three lines of choices again. The bottom line can only be true if the previous two choice lines end up false. This means that there are two Zeds which include the bottom line. The reason this approach works is that the top line is most selective, the second line is slightly less selective and the bottom line is the least selective. If you cannot order the lines like this, the chances are that your rules are ambiguous and could do with tweaking or alternatively, you should split the group into two.


Microsoft Excel - OfficeCalculator.xls
File Edit View Insert Format Tools Data Window Help
view dependents
Calibri
11
Reply with Changes... End Review...
Security...
Description DataIn Logic DataOut
Ready
Start
Microsoft Excel - Offi...
Microsoft Excel Help
16:28

A8 =DataIn!B4="sales"

	A	B	C	D	E
1	Is Admin	Has Access > 9	Install 2007, L1		Rule Description
2	TRUE	TRUE	TRUE		Admin with later version Access
3	IsAdmin	Has Powerpoint	Install 2007,L2		
4	FALSE	FALSE	FALSE		Admin with Powerpoint
5	IsAdmin		Install 2007,L3		
6	FALSE		FALSE		Default Admin
7	IsSales	Has Powerpoint	Install 2007,L2		
8	FALSE	FALSE	FALSE		Sales with Powerpoint
9	IsSales		Install 2007,L3		
10	FALSE		FALSE		Default Sales
11	IsSupply	Has Word < 11	Install 2003		
12	FALSE	FALSE	FALSE		Sales, upgrade A
13	IsSupply	Has Excel < 11	Install 2003		
14	FALSE	FALSE	FALSE		Sales, upgrade B
15					
16		Install 2007, L1	TRUE		
17		Install 2007, L2	FALSE		
18		Install 2007, L3	FALSE		
19		Install 2003	FALSE		
20					

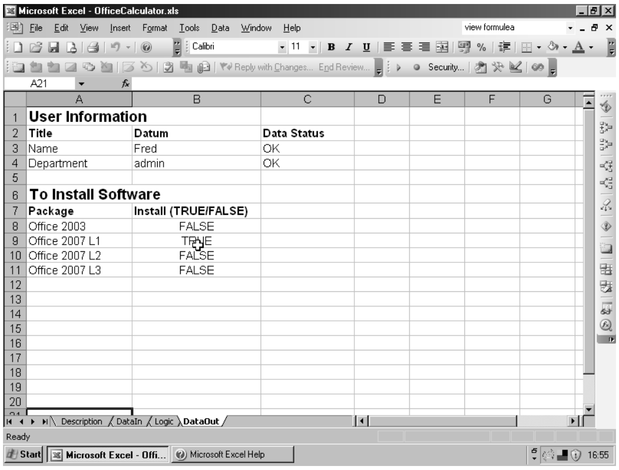
Figure 56: The next Block in our example is shown here with its Zeds revealed by Excel's Cell dependency tracking.



A12 =DataIn!B4="supply"

	A	B	C	D	E
1	**Is Admin**	**Has Access > 9**	**Install 2007, L1**		**Rule Description**
2	TRUE	TRUE	TRUE		Admin with later version Access
3	**IsAdmin**	**Has Powerpoint**	**Install 2007,L2**		
4	FALSE	FALSE	FALSE		Admin with Powerpoint
5	**IsAdmin**		**Install 2007,L3**		
6	FALSE		FALSE		Default Admin
7	**IsSales**	**Has Powerpoint**	**Install 2007,L2**		
8	FALSE	FALSE	FALSE		Sales with Powerpoint
9	**IsSales**		**Install 2007,L3**		
10	FALSE		FALSE		Default Sales
11	**IsSupply**	**Has Word < 11**	**Install 2003**		
12	FALSE	FALSE	FALSE		Sales, upgrade A
13	**IsSupply**	**Has Excel < 11**	**Install 2003**		
14	FALSE	FALSE	FALSE		Sales, upgrade B
15					
16		**Install 2007, L1**	TRUE		
17		**Install 2007, L2**	FALSE		
18		**Install 2007, L3**	FALSE		
19		**Install 2003**	FALSE		
20					

Figure 57: Here is the last of the Rule blocks with its internal structure displayed. The dotted dependency line near the bottom shows the way the result of the block is reported forward onto the "Data Out" Sheet.



	A	B	C
1	**User Information**		
2	**Title**	**Datum**	**Data Status**
3	Name	Fred	OK
4	Department	admin	OK
5			
6	**To Install Software**		
7	**Package**	**Install (TRUE/FALSE)**	
8	Office 2003	FALSE	
9	Office 2007 L1	TRUE	
10	Office 2007 L2	FALSE	
11	Office 2007 L3	FALSE	

Figure 58: In this example I have made a separate "Data Out" Sheet which derives its data from the rule sheet yet presents the results in an unambiguous way.

Moving From One To Many

So far in choice/decision processing we have seen an explanation and examples of processing for one input data set. In the previous example that data set represented one person. However, at the beginning of this section I discussed how this technique could be applied to large numbers of data sets.

It turns out to be straight forward to just take the work done so far and convert it to doing multiple data sets. We need not alter our approach to the choice/decision Workbook at all. The three Sheet structure (DataIn, Logic, DataOut) is ideal. The trick is to develop on this Workbook until it gives the correct results for every possible set of input data. Once this is done we leave it alone. The multiple data set system 'wraps' our decision Workbook in a script which does the following:

1) Open a Workbook with all the data sets, one per row in a Sheet. Call this Workbook *Input*.
2) Open the Logic Workbook. Call this Workbook *Decision*.
3) Open a Workbook for the recording of the decisions. Call this Workbook O*utput*.
4) For each row in *Input* take the data in that Row and place it in the DataIn Sheet in *Decision*.
5) For each loading of a data set into Decision, take the result from DataOut in Decision and place it in a Row in Output.

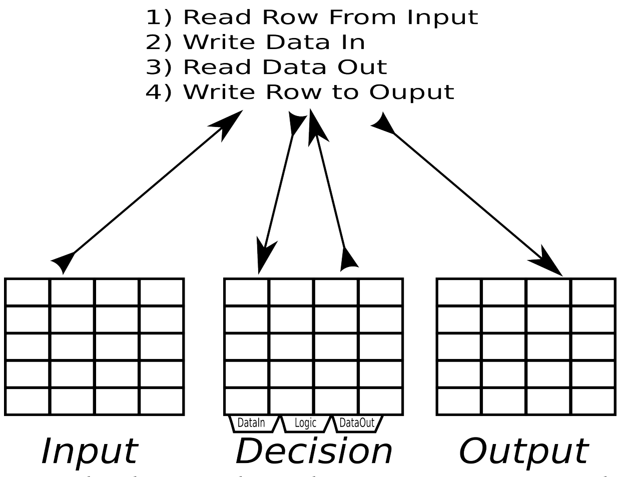


Figure 59: This diagram shows the way a script processes data to make a 'one at a time' decision Workbook turn into a powerful decision crunching system.

Lesson 10: Speeding Scripts Up

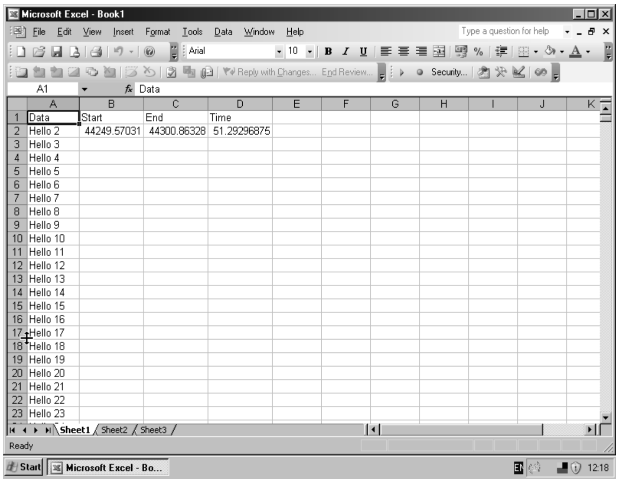
Cell By Cell Access Is Slow

The natural way to manipulate data in Excel from a Scripting Macro is one Cell at a time. The only problem with this approach is speed. It takes an appreciable amount of time to place data into a Cell. Setting the Value of a single Cell might be fast enough to seem instant to the human eye, however doing so for thousands of Cells can take seconds or even minutes.

To illustrate this issue, here is a script which loads one thousand Cells one at a time into a newly created Workbook.

```
set myExcel     = CreateObject("Excel.Application")
myExcel.Visible = true
set myWorkbook  = myExcel.Workbooks.Add
set mySheet     = myWorkbook.Sheets(1)

' Get the start time
startTime       = Timer()

' Load values into 1000 Cells directly
for row = 2 to 1001
mySheet.Cells(row,1).Value = "Hello " & row
next

' Get the end time
endTime = Timer

' Lay out the results column headings,
' Work out the result using an Excel formula
' and make the Columns automatically set themselves
' to an appropriate width
With mySheet
    .Cells(1,1).Value = "Data"
    .Cells(1,2).Value = "Start"
    .Cells(1,3).Value = "End"
    .Cells(1,4).Value = "Time"

    .Cells(2,2).Value   = startTime
    .Cells(2,3).Value   = endTime
    .Cells(2,4).Formula = "=C2-B2"

    .Columns(1).Autofit
    .Columns(2).Autofit
    .Columns(3).Autofit
    .Columns(4).Autofit
End With
```

The key part of the script is the For loop in the middle. It places a different String into each of the Cells in Column 1 from Row 2 to 10001. The VBScript 'built in' function *Timer()* is used to record the number of seconds since midnight before and after the loop has run. The difference in the two results is the time it took the loop to run. *Timer()* is usually accurate to within around twenty milliseconds, so it makes quite a handy way to time the performance of script.



	A	B	C	D
1	Data	Start	End	Time
2	Hello 2	44249.57031	44300.86328	51.29296875
3	Hello 3			
4	Hello 4			
5	Hello 5			
6	Hello 6			
7	Hello 7			
8	Hello 8			
9	Hello 9			
10	Hello 10			
11	Hello 11			
12	Hello 12			
13	Hello 13			
14	Hello 14			
15	Hello 15			
16	Hello 16			
17	Hello 17			
18	Hello 18			
19	Hello 19			
20	Hello 20			
21	Hello 21			
22	Hello 22			
23	Hello 23			

Figure 60: This is a screen shot showing the result of running the Cell by Cell loading script. It shows that it has taken 51.3 seconds to load a String into 1000 Cells when the data is loaded on Cells at a time.

As we can see from Figure 60, it has taken nearly a minute to load one thousand Cells. If we were to do this for ten thousand Rows over ten Columns, then it would take around one hundred minutes. If we imagine running a script that takes over an hour, then finding something has gone wrong and having to run it again, it is easy to see why speeding things up is good!

Fortunately, there are several ways to speed up Scripting Macros. A good approach is often to develop the script using a small set of data and loading Cells one at a time. Once it is working correctly it is time to start speeding it up using the techniques covered here.

Bulk Data Loading With Arrays

What Is An Array?

An Array is a special sort of value for a Variable. We have already seen that Variables can hold things like String, Object, Date etc. What happens if we want one Variable to hold lots of Strings or lots of Objects. This is where Arrays come in.

To hold three Strings we could put:

```
s1 = “hello 1”
s2 = “hello 2”
s3 = “hello 3”
```

Or we could create a Variable using the VBScript '*dim*' special word:

```
Dim myString(2)
myString(0) = “hello 1”
myString(1) = “hello 2”
myString(2) = “hello 2”
```

That piece of code has a few things about it which could be confusing. Rather the pretend that it does not, let us tackle it head on.

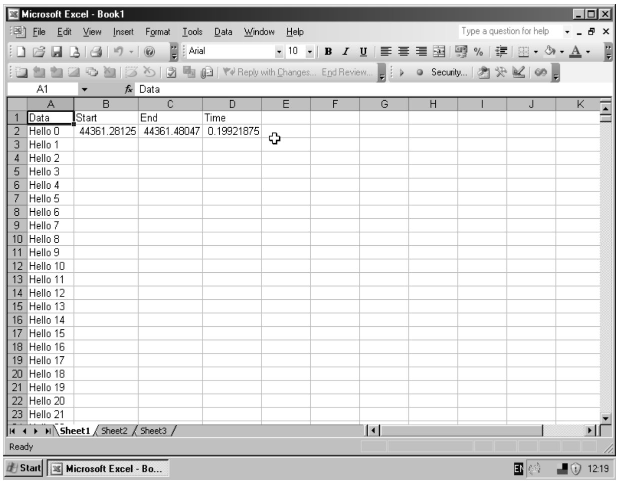
1. Why do we dim myString(2) when we want 3 elements in our array?
2. Why do we start at myString(0)?
3. I thought putting parenthesis after something meant it is a function.

When we create an Array we do not directly tell VBScript how many elements it should have; we tell VBScript what the 'upper bound' is. The 'lower bound' is zero by default. So, *Dim myString(2)* says “make an Array from 0 to 2”. This answers questions 1 and 2. Question 3 is answered by history. VBScript has it origins in a language called “Beginner's All Purpose Symbolic Instruction Code” which is rather old nowadays. More recent languages have removed ambiguity between Function and Array notation. As we are stuck with VBScript, we have to remember what names denote Arrays and which denote Functions. this can be made easier by always using a capital first letter for Functions and lower case for all Variables including Arrays.

Using Arrays Like Spread Sheets

```
set myExcel     = CreateObject("Excel.Application")
myExcel.Visible = true
set myWorkbook  = myExcel.Workbooks.Add
set mySheet     = myWorkbook.Sheets(1)

' Get the start time
startTime       = Timer()
```

```
  ' Create a 1000 by 1 array and load data into it
  dim data(999,0)
  for row = 0 to 999
    data(row,0) = "Hello " & row
  next

  ' Bulk load a 1000 by 1 Cells Range of the spread
  ' sheet from the array
  mySheet.Range _
  ( _
    mySheet.Cells(2,1), _
    mySheet.Cells(1002,1) _
  ).Value=data

  ' Get the end time
  endTime=Timer

  ' Here I have omitted the result printing and
  ' formating because it is identical to the previous
  ' script.
```



	A	B	C	D
1	Data	Start	End	Time
2	Hello 0	44361.28125	44361.48047	0.19921875
3	Hello 1			
4	Hello 2			
5	Hello 3			
6	Hello 4			
7	Hello 5			
8	Hello 6			
9	Hello 7			
10	Hello 8			
11	Hello 9			
12	Hello 10			
13	Hello 11			
14	Hello 12			
15	Hello 13			
16	Hello 14			
17	Hello 15			
18	Hello 16			
19	Hello 17			
20	Hello 18			
21	Hello 19			
22	Hello 20			
23	Hello 21			

Figure 61: This is a screen shot showing the result of running the Array to Cell loading script. It shows that it has taken less than a fifth of a second to load a String into 1000 Cells when the data is loaded on into an Array and then the Array bulk loaded into the Cells. This is 250 times faster than doing the load one Cell at a time.

Bulk Data Transfer With Copy/Paste

When we move data around by hand we use Copy and Paste all the time. It is very much faster that Cell by Cell transfer of values and so is ideal for scripting. Here is an example:

```
Option Explicit
Dim myExcel,myWorkbook,mySheet
Set myExcel=CreateObject("Excel.Application")
myExcel.Visible=TRUE
Set myWorkbook=myExcel.Workbooks.Add()
Set mySheet=myWorkbook.Sheets(1)
mySheet.Cells(1,1).Value="Hello"
' Please note that we must make sure the Sheet from
' which we Copy and that to which we Paste are the
' active (visible) ones.  This is done via the
' Sheet.Select Method.
mySheet.Select
mySheet.Cells.Copy
myWorkbook.Sheets(2).Select
myWorkbook.Sheets(2).Paste
```

Don't Forget Formulae And Fills

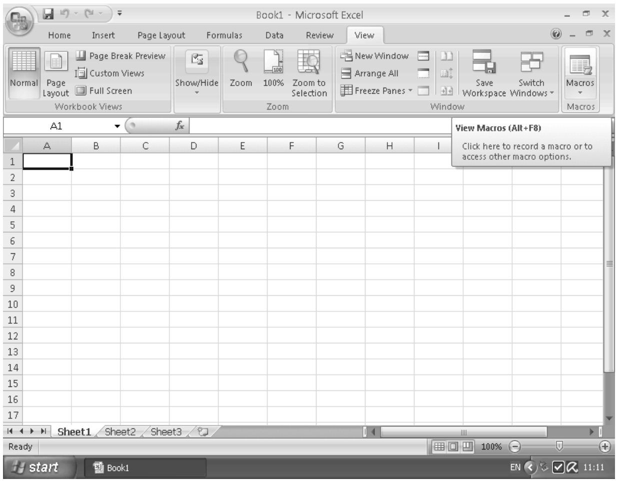
We have seen using fills before in lesson 8. It is much faster to fill a Column with a formula than to calculate Cell values one by one. Here is an example:

```
mySheet.Cells(1,5).Value = "Over Limit"
mySheet.Cells(2,5).Formula = "=if(C2>14,1,0)"
mySheet.Range("E2:E301").FillDown
```

Appendix A: Differences With Excel 2007

Recording Macros:

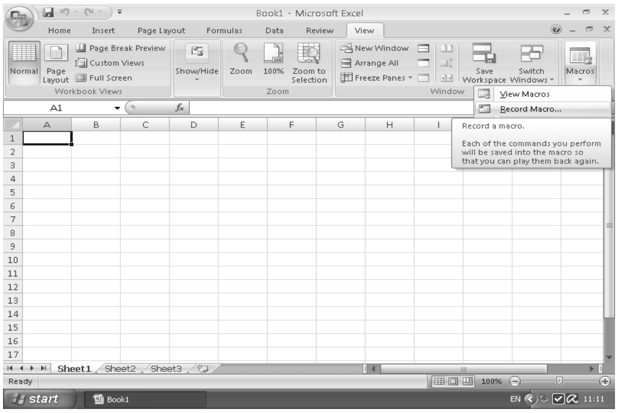
Recording Macros is pretty much exactly the same in Excel 2007. The only real differences relate to finding the options in Microsoft's new layout. Here are some screen shots showing the basic steps. The concepts are identical to those described in lesson 4.



View Macros (Alt+F8)

Click here to record a macro or to access other macro options.

Figure 62: To start recording, we must click the 'Macro' pull down under the 'View' ribbon.



Record a macro.

Each of the commands you perform will be saved into the macro so that you can play them back again.

Figure 63: Clicking the 'Macro' pull down gives us two options. To record a Macro, we just click 'Record Macro'. Selecting the Macro name etc. is just the same as for previous versions of Excel.

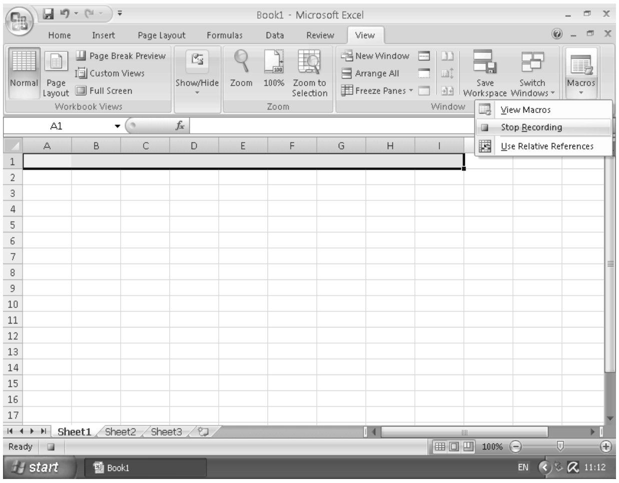
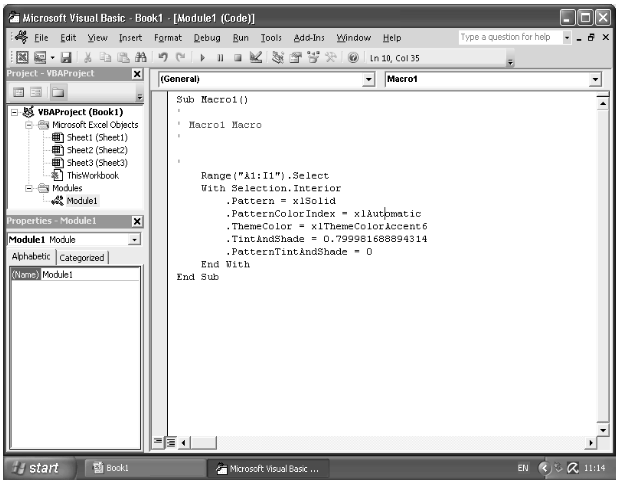


Figure 64: Once we finished performing the actions which we want recorded, the 'View/Macro' pull down gives us the option to 'Stop Recording'.

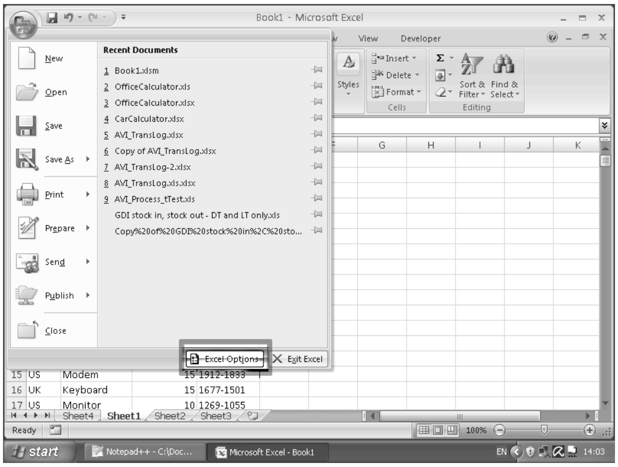


```
Sub Macro1()
'
' Macro1 Macro
'

'
    Range("A1:I1").Select
    With Selection.Interior
        .Pattern = xlSolid
        .PatternColorIndex = xlAutomatic
        .ThemeColor = xlThemeColorAccent6
        .TintAndShade = 0.799981688894314
        .PatternTintAndShade = 0
    End With
End Sub
```

Figure 65: The “View/Macro” pull down lets us view the code of the Macro. The whole of the code editor is petty much identical to that of previous versions of Excel.

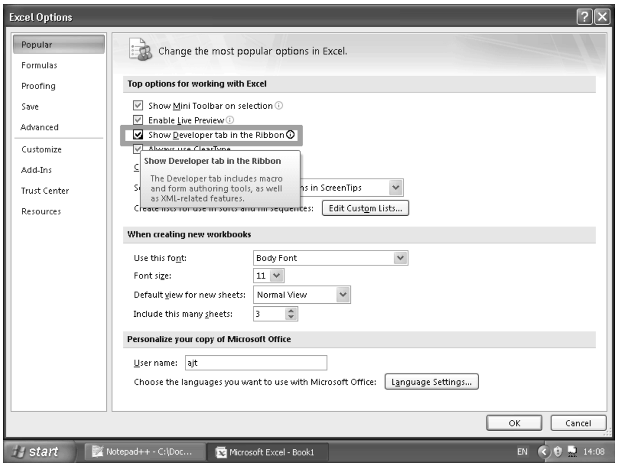
Showing The Developer Tab (Accessing VBA)

It would appear that the default install of Excel 2007 hides the Developer Tab. To be able to see it, and so gain easy access to the Code Editor and Object Browser, we can follow these steps.



15	US	Modem	15	1912-1833
16	UK	Keyboard	15	1677-1501
17	US	Monitor	10	1269-1055

Figure 66: Step one is to click on the MS Office symbol in the top left hand corner of Excel's window. Doing so reveals the menu shown above. At the bottom right is a button (highlighted in the picture) named "Excel Options". This must be clicked.



Change the most popular options in Excel.

Top options for working with Excel

- [✓] Show Mini Toolbar on selection
- [✓] Enable Live Preview
- [✓] Show Developer tab in the Ribbon

Show Developer tab in the Ribbon

The Developer tab includes macro and form authoring tools, as well as XML-related features.

Create lists for use in sorts and fill sequences:

When creating new workbooks

Use this font: Body Font

Font size: 11

Default view for new sheets: Normal View

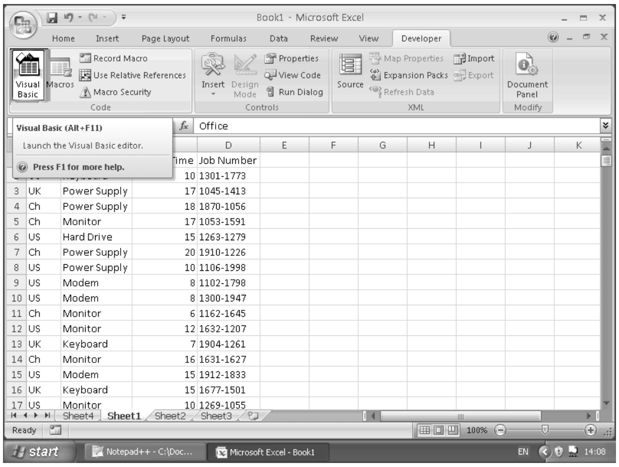
Include this many sheets: 3

Personalize your copy of Microsoft Office

User name: ajt

Choose the languages you want to use with Microsoft Office:

Figure 67: Clicking the "Excel Options" button reveals the Excel Options window. Once one has recovered from the shock, it becomes visible that there is an option "Show Developer Tab in the Ribbon". The trick is to select this and then click "OK".



Visual Basic (Alt+F11)

Launch the Visual Basic editor.

Press F1 for more help.

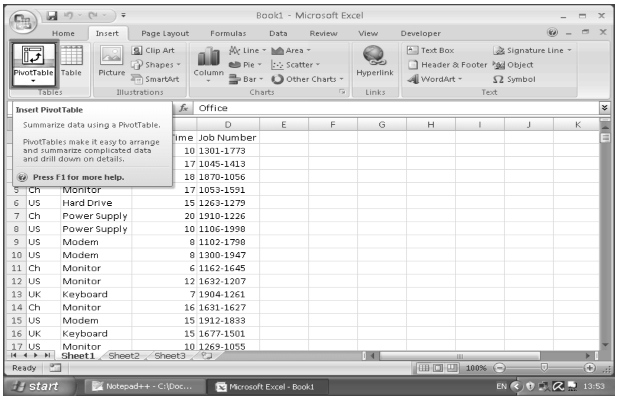
Office

			D	E	F	G	H	I	J	K
			ime	Job Number						
			10	1301-1773						
3	UK	Power Supply	17	1045-1413						
4	Ch	Power Supply	18	1870-1056						
5	Ch	Monitor	17	1053-1591						
6	US	Hard Drive	15	1263-1279						
7	Ch	Power Supply	20	1910-1226						
8	US	Power Supply	10	1106-1998						
9	US	Modem	8	1102-1798						
10	US	Modem	8	1300-1947						
11	Ch	Monitor	6	1162-1645						
12	US	Monitor	12	1632-1207						
13	UK	Keyboard	7	1904-1261						
14	Ch	Monitor	16	1631-1627						
15	US	Modem	15	1912-1833						
16	UK	Keyboard	15	1677-1501						
17	US	Monitor	10	1269-1055						

Figure 68: Here we can see our new Developer Tab . Highlighted in this screen shot is the Visual Basic button. Clicking on this opens the Code Editor. The Code Editor and all its options (e.g. the Object Browser) are more or less identical to those of earlier versions of Excel.

Creating A Pivot Table

The concepts behind creating a Pivot Table in Excel 2007 are identical to those for previous versions. Never the less, the steps involved in doing so are slightly different and the wizard has changed considerably. I have shown the steps involved using the following series of screen shots with captions:



Insert PivotTable

Summarize data using a PivotTable.

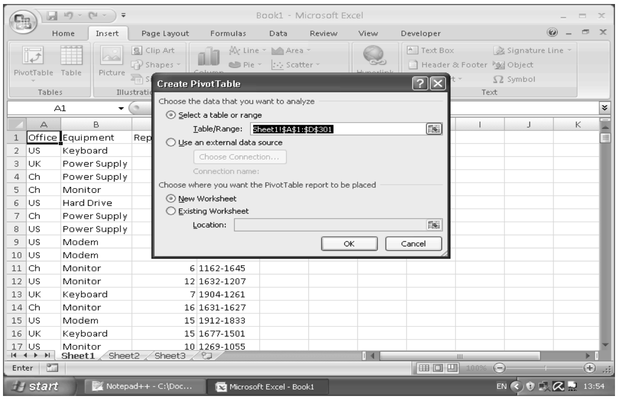
PivotTables make it easy to arrange and summarize complicated data and drill down on details.

Press F1 for more help.

Office

				D
1			ime	Job Number
2			10	1301-1773
3			17	1045-1413
4			18	1870-1056
5	Ch	Monitor	17	1053-1591
6	US	Hard Drive	15	1263-1279
7	Ch	Power Supply	20	1910-1226
8	US	Power Supply	10	1106-1998
9	US	Modem	8	1102-1798
10	US	Modem	8	1300-1947
11	Ch	Monitor	6	1162-1645
12	US	Monitor	12	1632-1207
13	UK	Keyboard	7	1904-1261
14	Ch	Monitor	16	1631-1627
15	US	Modem	15	1912-1833
16	UK	Keyboard	15	1677-1501
17	US	Monitor	10	1269-1055

Figure 69: Kicking off creating a new Pivot Table comes from clicking the "Pivot Table" button on the Insert Tab of the Ribbon.



Create PivotTable

Choose the data that you want to analyze

[✓ Select a table or range]

Table/Range: Sheet1!$A$1:$D$301

Use an external data source

Choose Connection...

Connection name:

Choose where you want the PivotTable report to be placed

[✓ New Worksheet]

Existing Worksheet

Location:

OK Cancel

	A	B		
1	Office	Equipment	Rep	
2	US	Keyboard		
3	UK	Power Supply		
4	Ch	Power Supply		
5	Ch	Monitor		
6	US	Hard Drive		
7	Ch	Power Supply		
8	US	Power Supply		
9	US	Modem		
10	US	Modem		
11	Ch	Monitor	6	1162-1645
12	US	Monitor	12	1632-1207
13	UK	Keyboard	7	1904-1261
14	Ch	Monitor	16	1631-1627
15	US	Modem	15	1912-1833
16	UK	Keyboard	15	1677-1501
17	US	Monitor	10	1269-1055

Figure 70: The Pivot Table wizard starts with the pop up shown above. I highly recommend that "New Worksheet" is selected. Most of the time Excel will automatically guess the correct range of Cells from which to take the data for our Pivot Table. If it does not, we can directly type the range into the "Table Range" text box. And alternative is to select a range of Cells by dragging the mouse over the Sheet.

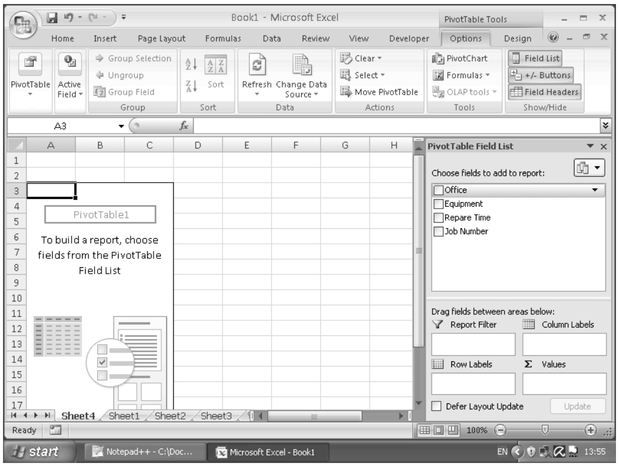


Figure 71: Once we have clicked "OK" on the first pop-up, Excel inserts a blank Pivot Table and the "Pivot Table Field List" box as shown in this screen shot. This is really very different from the previous versions of Excel. We no longer see areas in the blank Pivot Table for drag-and-drop. Constructing the table is all done via the Pivot Table Field List box.

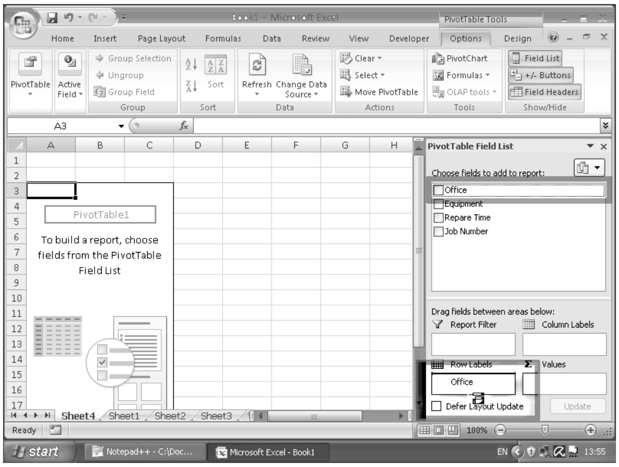
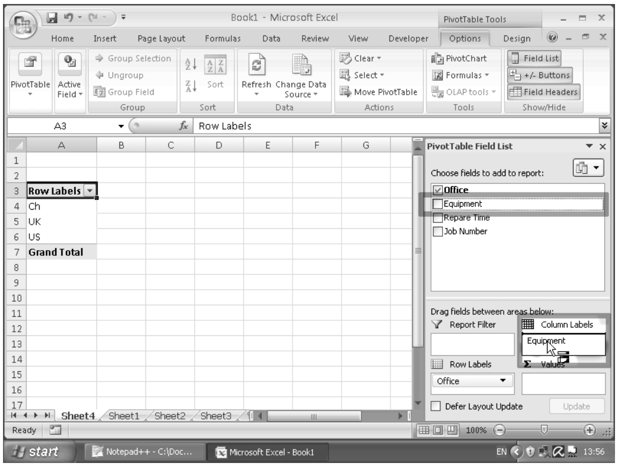
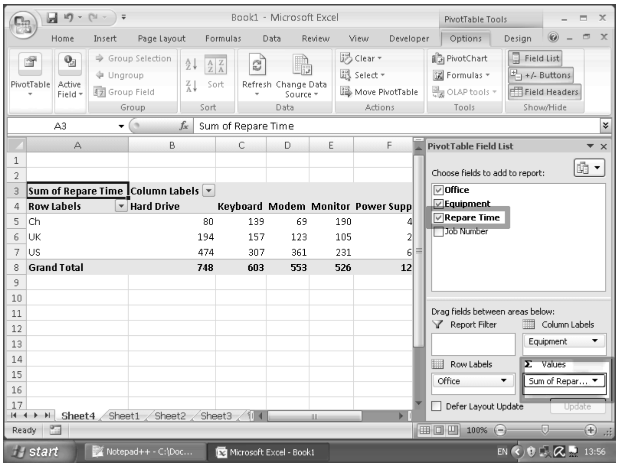


Figure 72: To summarise a field across rows, we drag the field name from the top pane of the Pivot Table Field List box to the "Row Labels" area.



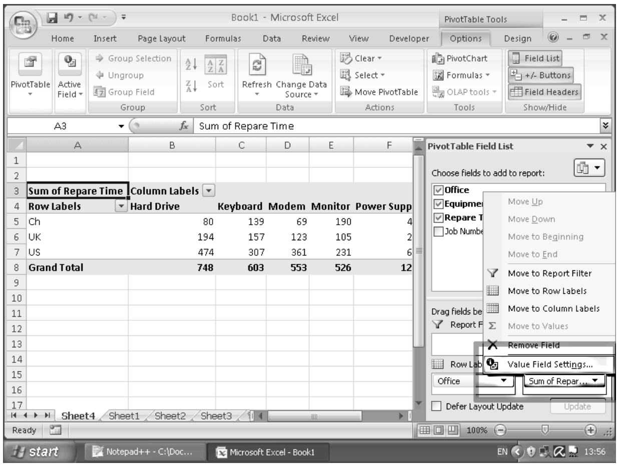
A
Row Labels
Ch
UK
US
Grand Total

Figure 73: To summarise a field down Columns, we drag-and-drop the field name from the top pane to the "Column Labels" area.



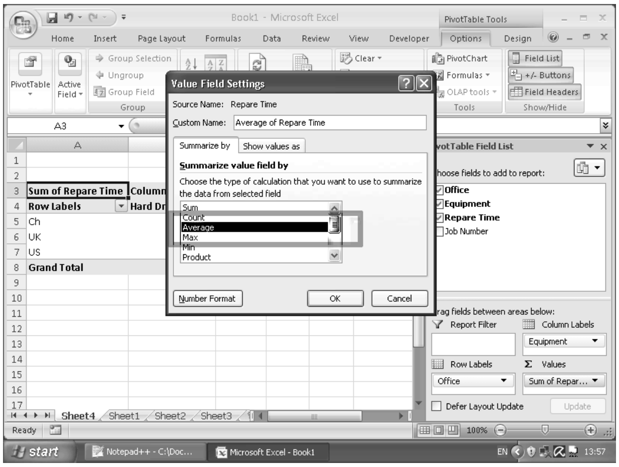
Sum of Repare Time	Column Labels				
Row Labels	Hard Drive	Keyboard	Modem	Monitor	Power Supp
Ch	80	139	69	190	4
UK	194	157	123	105	2
US	474	307	361	231	6
Grand Total	748	603	553	526	12

Figure 74: In the bottom right of the Pivot Table Field List box is where the Data field names are dragged. This is highlighted in the above screen shot. It starts out with the default "Sum" function. To set up the data which we want summarising we first drag that field name to this area. Just as with the previous version of Excel, changing to summary method it something other than "Sum" is a separate step (described next).



Sum of Repare Time	Column Labels				
Row Labels	Hard Drive	Keyboard	Modem	Monitor	Power Supp
Ch	80	139	69	190	4
UK	194	157	123	105	2
US	474	307	361	231	6
Grand Total	748	603	553	526	12

Figure 75. Right clicking on the Data area of the Pivot Table Field List box brings up the context menu shown above. At the bottom of this menu is the "Value Field Settings" option (highlighted). Clicking this reveals the options necessary to change the summary method (next).



Value Field Settings

Source Name: Repare Time

Custom Name: Average of Repare Time

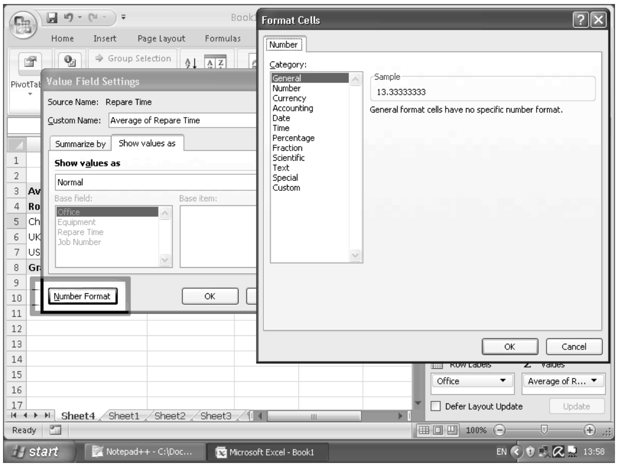
Summarize by | Show values as

Summarize value field by

Choose the type of calculation that you want to use to summarize the data from selected field

Sum
Count
Average
Max
Min
Product

Figure 76: We now get the "Value Field Settings" box. In the screen shot above, I am choosing "Average" for my Data fields (which Microsoft now sometimes call Value fields, but relate to the Data Area of previous versions of Excel).


Value Field Settings

Source Name: Repare Time

Custom Name: Average of Repare Time

Show values as: Normal

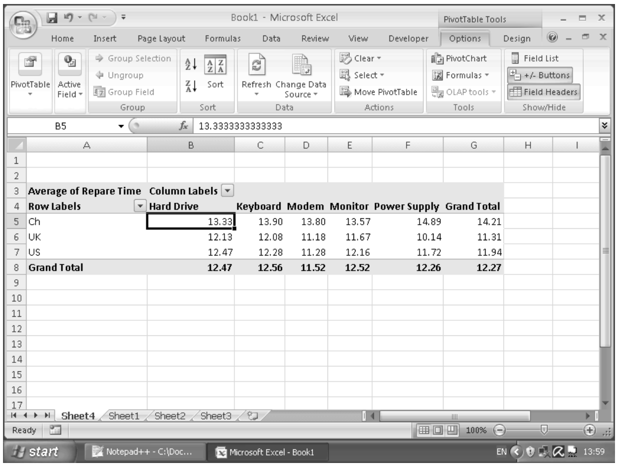
Format Cells

Category: General

Sample: 13.33333333

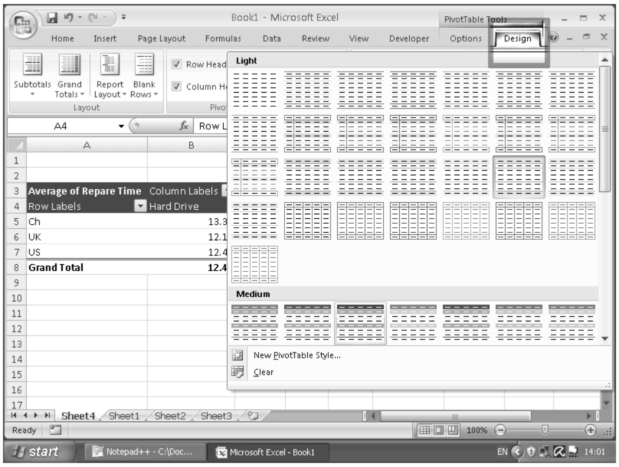
General format cells have no specific number format.

Figure 77: "Value Field Settings" box also has a tab for "Show values as". Under this tab we can find a button (highlighted) which brings up the "Format Cells" box for the Pivot Table. This can be used to set the number format. In this example I have set it to 2 decimal places (next).


B5 = 13.3333333333333

Average of Repare Time	Column Labels					
Row Labels	Hard Drive	Keyboard	Modem	Monitor	Power Supply	Grand Total
Ch	13.33	13.90	13.80	13.57	14.89	14.21
UK	12.13	12.08	11.18	11.67	10.14	11.31
US	12.47	12.28	11.28	12.16	11.72	11.94
Grand Total	**12.47**	**12.56**	**11.52**	**12.52**	**12.26**	**12.27**

Figure 78: Here we can see our new Pivot Table is default layout with a 2 decimal place number format.



Average of Repare Time	Column Labels
Row Labels	Hard Drive
Ch	13.3
UK	12.1
US	12.4
Grand Total	12.4

Figure 79: By using the "Design" tab (highlighted) we can pick one of the many built in layouts for our table if the default is not quite as visually appealing as we want.

In lesson 8 we looked at the 'raw' Macro for creating a Pivot Table just as Excel had recorded it. For interests sake, here is the raw Macro which was recorded for the creation the the Pivot Table in the screen shots here. There is very little difference. As usual, a Macro set up to run for an earlier version of Excel will (probably) work in Excel 2007, but we should not expect the reverse to be true:

```
  ' Raw Macro Recording For Creating A Pivot Table In
Excel 2007
    Sheets.Add
    ActiveWorkbook.Worksheets("Sheet4").PivotTables("Piv
otTable1").PivotCache. _
      CreatePivotTable TableDestination:="Sheet5!R3C1",
TableName:="PivotTable1" _
      , DefaultVersion:=xlPivotTableVersion12
    Sheets("Sheet5").Select
    Cells(3, 1).Select
    ActiveWorkbook.ShowPivotTableFieldList = True
    With
ActiveSheet.PivotTables("PivotTable1").PivotFields("Offi
ce")
      .Orientation = xlRowField
      .Position = 1
    End With
    With
ActiveSheet.PivotTables("PivotTable1").PivotFields("Equi
pment")
      .Orientation = xlColumnField
      .Position = 1
    End With
    ActiveSheet.PivotTables("PivotTable1").AddDataField
ActiveSheet.PivotTables( _
      "PivotTable1").PivotFields("Repair Time"), "Sum of
Repair Time", xlSum
    With
ActiveSheet.PivotTables("PivotTable1").PivotFields("Sum
of Repair Time")
      .Caption = "Average of Repair Time"
      .Function = xlAverage
    End With
    With
ActiveSheet.PivotTables("PivotTable1").PivotFields( _
      "Average of Repair Time")
      .NumberFormat = "0.00"
    End With
    ActiveSheet.PivotTables("PivotTable1").TableStyle2 =
"PivotStyleLight19"
```

Appendix B: How This Book Was Written

I figured it might be interesting to some readers of this book to know the methodology which went into writing it. Well, it might come as something of a shock. This book was written on a Linux computer! The software I used was as follows:

- Kubuntu 7.10 (Linux operating system)
- Open Office Writer 2.3 (Word processor)
- The GIMP 2.4.0-rc3 (Graphics manipulation and drawing)
- Ksnapshot 0.7 (Screen capture)
- Qemu 0.9 (Virtual Machine for running Windows on Linux)
- Inkscape 0.45 (Drawing package for diagrams)
- KPDF (Previewing PDF copies)
- Filezilla 3.0.0 (File transfers on and off the virtual machine)
- Terminal Sever Client 0.148 (Viewing remote Windows machine)

The basic process of creating the book was to:

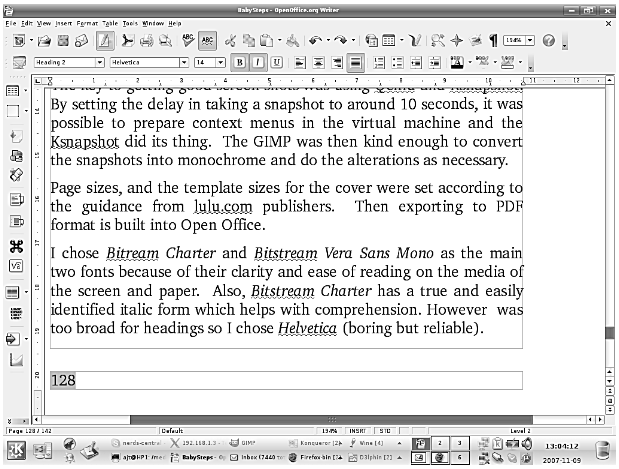
1) Write sections of it long hand.
2) Go do the scripts and screen shots for the section.
3) Type it up and stick it all together into Open Office.
4) Read and correct (lots).
5) Print out.
6) Add stuff long hand to the printouts.
7) Repeat as necessary.

The key to getting good screen shots was using Qemu and Ksnapshot. By setting the delay in taking a snapshot to around 10 seconds, it was possible to prepare context menus in the virtual machine and the Ksnapshot did its thing. The GIMP was then kind enough to convert the snapshots into monochrome and do the alterations as necessary.

Page sizes, and the template sizes for the cover were set according to the guidance from lulu.com publishers. Then exporting to PDF format is built into Open Office.

I chose *Bitream Charter* and *Bitstream Vera Sans Mono* as the main two fonts because of their clarity and ease of reading on the media of the screen and paper. Also, *Bitstream Charter* has a true and easily identified italic form which helps with comprehension. One final good feature of the font is that is narrow, meaning that Baby Steps does not have to kill too many trees to be printed!

I personally I prefer non serif fonts for headings, especially where the body text has serifs. Unfortunately, *Bitstream Vera Sans* was too broad for headings so I chose *Arial* (boring but reliable). Also Arial makes a nice heading font because it is clear and open so is easily read without dominating the page.


BabySteps - OpenOffice.org Writer
File Edit View Insert Format Table Tools Window Help
Heading 2
Helvetica
14
194%
Page 128 / 142
Default
INSRT
STD
Level 2
13:04:12
2007-11-09

By setting the delay in taking a snapshot to around 10 seconds, it was possible to prepare context menus in the virtual machine and the Ksnapshot did its thing. The GIMP was then kind enough to convert the snapshots into monochrome and do the alterations as necessary.

Page sizes, and the template sizes for the cover were set according to the guidance from lulu.com publishers. Then exporting to PDF format is built into Open Office.

I chose *Bitream Charter* and *Bitstream Vera Sans Mono* as the main two fonts because of their clarity and ease of reading on the media of the screen and paper. Also, *Bitstream Charter* has a true and easily identified italic form which helps with comprehension. However was too broad for headings so I chose *Helvetica* (boring but reliable).

128

Figure 80: Here is me, on my Kubuntu laptop, writing the section of Baby Steps on how Baby Steps was written.

Appendix C: Why Script When Excel Has Macros?

In “Baby Steps” I have been showing how to write scripts which open Excel application and manipulate Excel spreadsheets through these applications. One obvious questions is why bother with scripts, why not use Excel's built in Macro system.

There are several reasons which I will discuss separately. However, they all tend to relate to separation of data and code. An excel workbook with Macros attached has no separation of data and code; where ever the code goes, the data goes with it. This simple problem has myriad repercussions.

To go over there repercussions and thus the reasons for scripting, I shall use the format of formal argument. I present a question and answer session where I try to convince myself that scripting is better.

Q: First off, why bother at all. If I have a spreadsheet ad I want to automate processing on it, I can use a Macro.

A: Consider the following example. An automated data export from a database creates a comma separated value file each morning. You want that loaded into Excel and processed as soon as it turns up. Because a Macros has to be attached to the data, and you have a new data file each day, a Macro is no use to you.

Q: I could put the Macros into an empty Workbook and load the data into that each day.

A: Yes, that would work. However, you would have to ensure that your empty Workbook was not overwritten with the new filled Workbook each time. This would have to include ensuring it did not happen even if Excel crashed half way through working etc.

Q: OK, I can do that! How about the Macro containing Workbook automatically saves its self as a new document then loads the data, processes it and saves its self again.

A: Are, that might work quite well. However, that is a very complex approach. To make it functions in the field you will need to run distribute the Macro only book. Each time it runs, people will have to permit its Macros to run by hand, or you are going to have to digitally sign the Workbook and then get people to accept the certificate. Under some conditions, the users will not even be able to enable the Macros because the security level of their Excel will be locked to high and they will lack the permissions to reduce it.

Effectively, by the time you have an Excel Macro based solution, it is more complex and unwieldy that the script based one.

Q: But the VBA environment is so nice and easy to use in Excel, and it also has the Macro Recorder.

A: Yes, the VBA environment is well designed. However, a professional grade text editor like the free notepad++ is also very good. Notepad++ has some disadvantages over VBA's editor, but also some advantages like excellent find and replace, tabbed editing etc.

The other big advantage when it comes to environment is that scripts are text files and text files are easier to work with that embedded Macros. For example, you can use free tools like win-merge to find the differences between two copies of a script. It is easier to 'cut and paste' bits of code between scripts and scripts take up massively less space than Excel files.

When it comes to the Macro Recorder, that is a very rough tool. The be able to release a Recorded Macro into the field it need a lot of polishing. This means that once have Recorded your Macro, you will have to do some editing on it. Given this, you can just as easily copy it over into notepad++ and edit there as a script.

The other 'cool tool' in the Excel VBA environment is the Object Explorer. There is nothing to stop you still using it along with anything else (help for example) that comes with Excel.

Q: What about intellectual property protection? I can lock Macros with a password. Scripts are just text files.

A: MS produce a free program, the script encoder. The output is at least as secure as password protected Excel Macros.

One Final Thing

With MS Office 2007, the default file format for Excel is xlsx. This format does not even support embedded Macros. This indicates how the world in moving away from the idea of embedding code into documents.

Appendix D: VBScript Built In Functions & Constants

Built In Functions

VBScript has many built in functions. We have seen some of them being used in Baby-Steps so far. Here is a definitive list of them all and descriptions of what they do. I have added into the list the how often it is my experience that a particular built in function is actually used in Excel Marco Scripting. It is well worth trying to remember those which are used frequently.

The Functions Enumerated And Explained

Abs (rare)

Gives the absolute value of a number. For example Abs(-1) gives 1 whilst Abs(1) also gives one.

Array (rare)

Gives a Variant containing an array.

Asc (sometimes)

Gives the ANSI character code corresponding to the first letter in a string. In chapter 6 we looked at how files are lists of numbers and that by making different numbers represent different characters, we could encode text into files. Well, Asc returns the number with which a character will be encoded.

Atn (almost never)

Gives the arctangent of a number. I have been working in IT for twenty seven years and have never used this function in any computer language, includin

CBool (sometimes)

Tries to convert what we pass to it into a Boolean (TRUE/FALSE) value. Cbool(“TRUE”) gives true.

CByte (almost never)

Gives an expression that has been converted to a Variant of subtype Byte. The use of this is beyond most things one might do in Excel Macro Scripting.

CCur (almost never)

Gives an expression that has been converted to a Variant of subtype Currency. The use of this is beyond most things one might do in Excel Macro Scripting.

CDate (often)

Takes what ever we pass to it and tries to convert it into a Date. For example, CDate("10 January 2007") will give a date of the tenth of January two thousand and seven if the computer is set to work in English.

CDbl (sometimes)

This is handy for converting Strings into numbers where the Strings are being read from a file for example.

Chr (sometimes)

This is the logical opposite of Ansi. It converts a number into a character using the type of encoding that files use to store text. Chr(Ansi("A")) is "A".

CInt (almost never)

Use Cdbl or CLng instead.

CLng (rare)

This is handy for converting Strings into numbers where the Strings are being read from a file for example. However, most of the time CDbl is a better choice unless for some odd reason integer (non decimal) arithmetic is essential.

Cos (rare)

Gives the cosine of an angle. This is handy if you are working in engineering etc.

CreateObject (All the time).

This is one of the first things we looked at. For example CreateObject("Excel.Application")

CSng (almost never)

The use of this is beyond most things one might do in Excel Macro Scripting.

CStr (rare)

This converts whatever you pass to it into a String. One might think that we would use this all the time; however the & operator is even more useful and does much the same thing. For example, `CStr(1.23)` is the same as `"" & 1.23`.

Date (sometimes)

Gives the current system date.

DateAdd (often)

Use to add time (days, seconds etc.) to dates. See chapter 7. To use put DateAdd(interval,number,myDate) where the interval is one from this selection:

SETTING	DESCRIPTION
YYYY	Year
Q	Quarter
M	Month
Y	Day Of Year
D	Day
W	WeekDay
WW	Week Of Year
H	Hour
N	Minute
S	Second

So to add one week to the first of March two thousand and six we could put DateAdd("D",7,DateSerial(2006,3,1)).

DateDiff (sometimes)

This function works out the difference between two dates and returns the answer as a number of intervals. Those intervals are the same intervals as are used in DateAdd. To get the number of days between the first of March nineteen ninety seven and the current date, we could put DateDiff("D",DateSerial(1997,3,1),Date())

DatePart (rare)

Gives the specified part of a given date. It is called like this DatePart(interval, date).

DateSerial (often)

Creates a Date for a specified year, month, and day. See chapter 7. To create a Date representing the first of March nineteen ninety seven we would put DateSerial(1997,3,1).

DateValue (never)

Use CDate or DateSerial instead.

Day (rare)

Gives a whole number between 1 and 31, inclusive, representing the current day of the month.

Exp (rare)

Gives e (the base of natural logarithms) raised to a power.

Filter (rare)

Gives a zero-based array containing subset of a string array based on a specified filter criteria. The use of this is beyond the examples in this book.

Fix (rare)

Gives the integer portion of a number. If we put Wscript.Echo fix(1.5) we will get a popup box saying 1.

FormatCurrency (almost never)

Gives an expression formatted as a currency value using the currency symbol defined in the system control panel. Formatting like this is better done in Excel its self in Excel Macro Scripting.

FormatDateTime (almost never)

Gives an expression formatted as a date or time. Formatting like this is better done in Excel its self in Excel Macro Scripting.

FormatNumber (almost never)

Gives an expression formatted as a number. Formatting like this is better done in Excel its self in Excel Macro Scripting.

FormatPercent (almost never)

Gives an expression formatted as a percentage (multiplied by 100) with a trailing % character. Formatting like this is better done in Excel its self in Excel Macro Scripting.

GetObject (rare)

Gives a reference to a COM object from a file. This means that if the file has an 'binding' to a COM object (like .xls files are bound to Excel) then we can use GetObject to load the COM object and get the COM object to load the file. Frankly, this is a mess; please do not use it.

Hex (rare)

Gives a String representing the hexadecimal value of a number. Hex(32) gives “20” and Hex(255) gives “FF” etc.

Hour (rare)

Gives a whole number between 0 and 23, inclusive, representing the current hour of the day.

InStr (sometimes)

Gives the position of the first occurrence of one string within another. It also allows us to say where to start searching. So, Instr(“-”, “Alexander-Julian-Turner”) will give 10 and Instr(11,“-”, “Alexander-Julian-Turner”) will give 17.

InStrRev (rare)

Gives the position of an occurrence of one string within another, from the end of string. It is just like InStr but it works in reverse.

Int (almost never)

Gives the integer portion of a number. Use Clng instead.

IsArray (almost never)

Gives a Boolean (TRUE/FALSE) value specifying whether a Variable is an array.

IsDate (rare)

Gives a Boolean (TRUE/FALSE) value specifying whether an expression can be converted to a date. Sometimes we can get a value from a Sheet's Cell and it is handy to find out if Excel considered this value to be a Date. This built-in function will tell us.

IsEmpty (constant)

If a Cell or Variable has not ever had anything in it, then this will return TRUE otherwise it returns FALSE. Examples are IsEmpty(mySheet.Cell(row,1)) or IsEmpty(myVariable). We use IsEmpty to find empty rows in Sheets.

IsNull (sometimes)

Gives a Boolean (TRUE/FALSE) value that indicates whether an expression contains no valid data. If we work with data from databases then we will find these useful sometimes. It is probably a bit beyond the scope of Baby Steps.

IsNumeric (sometimes)

Gives a Boolean (TRUE/FALSE) value specifying whether an expression can be evaluated as a number.

IsObject (rare)

Gives a Boolean value specifying whether an expression references a valid Automation object. The only time I see this being used if Excel Scripting Macros is if something like a Word Document is embedded in a Sheet. Then the Value of a Cell would actually be an Object so IsObject(myCell.Value) might be used to detect this.

Join (sometimes)

Gives a string created by joining a number of substrings contained in an array. See chapter 11.

LBound (rare)

Gives the smallest available subscript for the specified dimension of an array.

LCase (often)

Gives a string that has been converted to lowercase

Left (sometimes)

Gives a specified number of characters from the left side of a string. Left("Dog Waffle", 3) will give "Dog".

Len (sometimes)

Gives the number of characters in a string or the number of bytes required to store a Variable. This is by far the most useful for Strings. For example Len("Dog Waffle") will give 10, i.e. the length of the String.

LoadPicture (almost never)

Gives a picture object.

Log (rare)

Gives the natural logarithm of a number.

LTrim (rare)

Gives a copy of a string without leading spaces. Normally we just use Trim().

Mid (often)

Read chunks out of Strings. Mid(“hello world”,6) gives “world” and Mid(“hello world”,6,2) gives “wo”.

Minute (rare)

Gives a whole number between 0 and 59, inclusive, representing the current minute of the hour.

Month (rare)

Gives a whole number between 1 and 12, inclusive, representing the current month of the year.

MonthName (rare)

Gives a string specifying the specified month. MonthName(1) will give “January” if the computer is set to use English. MonthName(1) will give “Janvier” if the computer is set to use French.

Now (often)

Gives the current date and time according to the setting of your computer's system date and time.

Oct (almost never)

Gives a string representing the octal value of a number.

Replace (often)

Gives a string in which a specified substring has been replaced with another substring a specified number of times. For example Replace(“10-10-2007”, “-”, “/”) will give “10/10/2007”. Replace(“10-10-2007”, “-”, “/”,1,1) will give “10/10-2007” and Replace(“10-10-2007”, “-”, “/”,4,1) will give “10-10/2007”.

Right (sometimes)

Gives a specified number of characters from the right side of a String.

Rnd (sometimes)

Gives a random number between 0 and 1.

Round (sometimes)

Gives a number rounded to a specified number of decimal places. For example Round(Rnd(),2) gives a two decimal place random number between 0 and 1 (e.g. 0.24).

RTrim (rare)

Gives a copy of a string without trailing spaces. We tend to use Trim() instead.

ScriptEngine (almost never)

Returns a string representing the scripting language in use.

ScriptEngineBuildVersion (almost never)

Returns the build version number of the script engine in use.

ScriptEngineMajorVersion (almost never)

Returns the major version number of the script engine in use.

Second (rare)

Gives a whole number between 0 and 59, inclusive, representing the current second of the minute

Sgn (rare)

Gives an integer specifying the sign of a number. If the number is negative, then this always give -1 otherwise it will give 1. Sgn(-0.0000123) gives -1 whilst Sgn(0) gives 1 and Sgn(131234) gives 1.

Sin (rare)

Gives the sine of an angle.

Space (sometimes)

Gives a string consisting of the specified number of spaces. Space(4) gives "   ".

Split (often)

Gives a zero-based, one-dimensional array containing a specified number of substrings. It is used Split(String-To-Split,String-Delimiter,Max-Number-Of-Pieces). myArray=Split("My Dog Is Cute", " ",3) will give a 3 element array starting with element 0. So from the array myArray(0) will have the value "My", myArray(1) will be

"Dog" and myArray(2) will be "Is Cute". If we miss out the Max-Number-Of-Pieces, it returns as many pieces as possible.

Sqr (rare)

Gives the square root of a number.

StrComp (rare)

Gives a value specifying the result of a String comparison.

String (sometimes)

Gives a repeating character string of the length specified. String(5,".") will give "....." .

StrReverse (almost never)

Gives a string in which the character order of a specified string is reversed. StrReverse("Dog") gives "goD".

Tan (rare)

Gives the tangent of an angle. This is notable only in that it is used more often than Atn!

Time (sometimes)

Gives a Date specifying the current system time.

TimeSerial (rare).

Gives a Date containing the time for a specific hour, minute, and second.

TimeValue (rare)

Gives a Date containing a time as worked out from a String etc. For example TimeValue("12:00:00 A.M.") or TimeValue("18:30:00").

Trim (constant)

Gives a copy of a string without leading and trailing spaces.

TypeName (rare)

Gives a string that provides type information about a Variable or Value. It can be useful for finding out what type a Cell Value is.

UBound (almost never)

Gives the largest available subscript for the specified dimension of an array.

UCase (sometimes)

Gives a string that has been converted to upper case. See LCase.

VarType (almost never)

Gives a value specifying the subtype of a Variable.

Weekday (rare)

Gives a whole number representing the current day of the week.

WeekdayName (rare)

Gives a string specifying the specified current day of the week.

Year (rare)

Gives a whole number representing current the year.

Built In Constants

VBScript built in constants can be use in place of Variables except you can only read from them you cannot assign to them. For example the following is OK:

```
dim myString
myString=vbcrlf
```

But this is not:

```
dim myString
myString = "abc"
vbcrlf="abc"
```

There are quite a few VBScript built in constants. To be completely honest, most are not really much use in Excel Macro Scripting; they are designed for other things using VBScript (like making web pages). However, the String constants are quite useful so I have described them here.

Built In String Constants

Constant	**Value**	**Description**
vbCr	Chr(13)	Carriage return.
VbCrLf	Chr(13) & Chr(10)	Carriage return–linefeed combination.
vbFormFeed	Chr(12)	Form feed, this is never really used nowadays (it relates to old style printers).
vbLf	Chr(10)	Line feed.
vbNewLine	Chr(13) &	Same as VbCrLf.

Constant	Value	Description
	Chr(10)	
vbNullChar	Chr(0)	Character having the value 0.
vbNullString	String having value 0	Not the same as a zero-length string (""), used for calling external procedures and not really in Excel Macro Scripting.
vbTab	Chr(9)	Horizontal tab, like pressing the 'tab' key.
vbVerticalTab	Chr(11)	Vertical tab, this is never really used nowadays (it relates to old style printers).

Appendix E: Named Arguments And The := Symbol

Why This Is Here

By all rights this section should be in Lesson 4. After all, it is about translation of recorded Macros into Scripting Macros. The problem being that there is already too much in Lesson 4. Not only that, the issue faced here is not commonly faced and requires more fiddling around than the rest of the translation stuff. So, I have it to an appendix.

The Where, Why And When Of :=

":=" is all about named arguments. If we consider a Method that instructs an Object to boil an egg. This Method will have arguments the tell it about how the egg should be boiled:

```
Boiler.BoilEgg(egg,hardness,coolAfter)
```

In my silly example the arguments are the egg to be boiled, how hard the egg should be and if the egg should be cooled down afterwards. Of these, the egg argument must always be passed to Boiler.BoilEgg; however, the hardness and coolAfter arguments could have default values. In other words, if we do not specify them, the Boiler just does the default. For example:

```
Boiler.BoilEgg(myEgg)
```

This might actually mean, boil myEgg, soft and cool afterwards. We only need to specify the other arguments if we do not want the default. So, to hard boil and cool afterwards we could put:

```
Boiler.BoilEgg(myEgg,xlHard)
```

To hard boil myEgg and not leave it to cool we could put:

```
Boiler.BoilEgg(myEgg,xlHard,FALSE)
```

The interesting bit comes if we want to soft boil the egg but not not cool it afterwards. In Excel Scripting Macros we would have to put something like this:

```
Boiler.BoilEgg(myEgg,xlSoft,FALSE)
```

Even though xlSoft is the default, we still have to specify it. This is because we need to specify the third argument (coolAfter) and the only way of doing so is to specify all three arguments. If might try to put:

```
Boiler.BoilEgg(myEgg,FALSE)
```

But this will cause an error. VBScript will not be able to tell that we meant FALSE for coolAfter; it will try and make the hardness FALSE which makes no sense at all.

Most of the time VBScript (which is used for Scripting Macros) and VBA (which is used by the Macro Recorder) as pretty much identical. However, there is a feature in VBA which could help with the egg boiling situation. VBA supports named arguments. So, in VBA one could actually put:

```
Boiler.BoilEgg(egg := myEgg, coolAfter := FALSE)
```

Why specifies only two of the three arguments but by using the argument names still works just fine. However, it is far from fine for us because we want to translate it into VBScript. This is a real world problem because the Macro Recorder sometimes uses named arguments which we have to translate into the simpler argument system of VBScript.

How To Translate From Named Arguments

Back in Lesson 8 we looked at creating Pivot Tables with a script. The raw recorded Macro looked something like this:

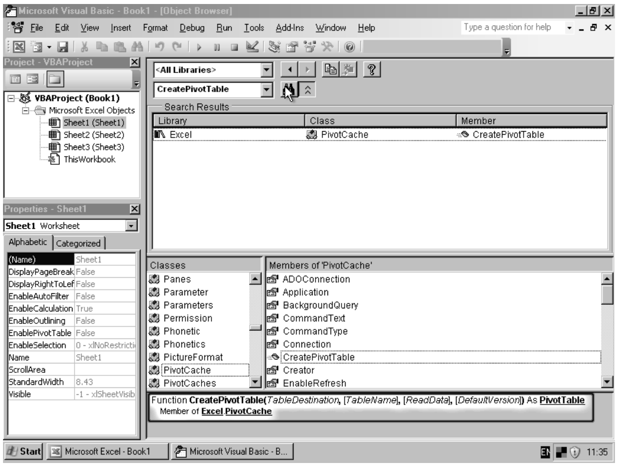
```
CreatePivotTable _
( _
  TableDestination:="", _
  TableName:= "PivotTable1", _
  DefaultVersion:=xlPivotTableVersion10 _
)
```

In the above code we can that the Macro Recorder as used named arguments for the CreatePivotTable Method. In general we can assume that it has created this in the correct order for using as non named argument. However, we cannot assume that it has specified all the arguments. In this case, simply removing the names will not work:

```
' This will fail!
CreatePivotTable _
( _
  "", _
  "PivotTable1", _
  xlPivotTableVersion10 _
  )
```

The solution is to use the Object Browser. We can "Search" for "CreatePivotTable" in the Object Browser. It will then show us what the full set of arguments is, and we can use that to correctly translate the recorded Macro into a Scripting Macro below:

```
  CreatePivotTable _
  ( _
    "", _
    "PivotTable1", _
    TRUE, _
    xlPivotTableVersion10 _
  )
```



Function CreatePivotTable(TableDestination, [TableName], [ReadData], [DefaultVersion]) As PivotTable
Member of Excel.PivotCache

Figure 81: Here I have searched for the definition of the CreatePivotTable Method. I have highlighted the definition in the picture. We can see that it takes 4 arguments rather than the 3 that the Macro Recorder supplied.

Errata

The code which generated the example for Pivot Tables (Lesson 8 and Appendix A) incorrectly used the words “repare time” which should have been “repair time”. Whilst the text of the book has been corrected, some of the screen shots show the incorrect spelling.

Index